Validation of CERN's Finite State Machines

Sander J.J. Leemans

8th June 2012

2IM91 - Master project

Graduation supervisor	Tim A.C. Willemse[a]
Graduation tutor	Jeroen J.A. Keiren[a]
Supervisor	Frank Glege[b]
Supervisor	Robert G.R. Garrido[b]

[a] Department of Mathematics and Computer Science,
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
[b] European Organization for Nuclear Research, CH-1211
Geneva 23, Switzerland

Abstract

CERN uses finite state machines organised in a hierarchical directed acyclic graph structure for its Detector Control Systems (DCSes) of the experiments of the Large Hadron Collider. These systems are huge and impossible to understand. We identified seven potential static semantic errors in the language used and located 182 errors in production systems; during the project 51 of these were repaired.

We improved existing tools to check systems for local loops and pairwise reachability of states, and provided the implementation to integrate the checks into the development environment. Moreover, we identified non-local loops as possible unwanted behaviour. For these non-local loops, we proved that, in some cases, it is necessary to traverse the complete state space of a system, which can be huge. Several reductions were introduced that reduce this state space. For one of the four large experiments, CMS, the state space was reduced from 10^{26402} to 10^{1189} and for ATLAS from 10^{59209} to 10^{35}. We identified a special kind of non-local loop that can be detected using bounded model checking. For the DCSes of three of the four experiments checking for these loops is feasible.

Using the tools we provided, the DCS developers can find errors and unwanted behaviour, repairing them increases confidence in the detector control systems.

Contents

1 Introduction

The *Large Hadron Collider* (LHC) at the European Organization for Nuclear Research (CERN) is a circular particle accelerator designed for the purpose of colliding particles in its experiments. It is built in its 27 kilometre circumference tunnel underneath France and Switzerland near Geneva. The *Compact Muon Solenoid* (CMS), *Large Hadron Collider Beauty* (LHCb), *A Toroidal LHC Apparatus* (ATLAS), and *A Large Ion Collider Experiment* (ALICE) experiments are the four big experiments at the LHC. CMS is designed to 'detect a wide range of particles and phenomena produced in the LHC's high-energy proton-proton and heavy-ion collisions'[1]. LHCb is an experiment set up to 'explore what happened after the Big Bang that allowed matter to survive and build the universe we inhabit today'[2]. ATLAS 'will explore the fundamental nature of matter and the basic forces that shape our Universe'[3]. ALICE is 'devoted to research in the physics of matter at an infinitely small scale'[4].

In all experiments, a *Detector Control System* (DCS) monitors and controls environment variables such as voltage, temperature and humidity within the detector. It consists of many pieces of hardware that measure and control over a million parameters concerning these environment variables. As the combination of all these sensors cannot be monitored by operators, a DCS also consists of software designed to a) gather information from its hardware sensors, b) summarise it to human operators and c) send commands to the hardware. Human operators monitor and control the DCS instead of the hardware.

The part of the software that we analysed performs the tasks b) and c). The architecture of this part of a DCS is shown in Figure 1.1: it consists of *nodes* organised in a hierarchical structure. Nodes are modelled as finite state machines. Intuitively, nodes gather the states of their children, use that information to move to a new state and notify their parents about the new state. Nodes also receive and send commands to trigger activity in their children. Nodes are modelled in the Finite State Machine language (FSM), being an abstraction of the State Manager Language (SML), which runs on the State Management Interface (SMI++) framework [FG05] that is part of the Joint Control Project (JCOP) framework [HGBGS05]. We introduce the term *CERN Finite State Machines* (CFSM) to denote the combination of FSM, nodes, their architecture and their implementation.

The complexity of single nodes is relatively low: on average a node contains about 8 states in CMS and ALICE and 6 states in LHCb and ATLAS. The

[1]http://cms.web.cern.ch/news/cms-experiment-cerns-lhc
[2]http://lhcb-public.web.cern.ch/lhcb-public/
[3]http://www.atlas.ch/glossary
[4]http://aliceinfo.cern.ch/Public/Welcome.html

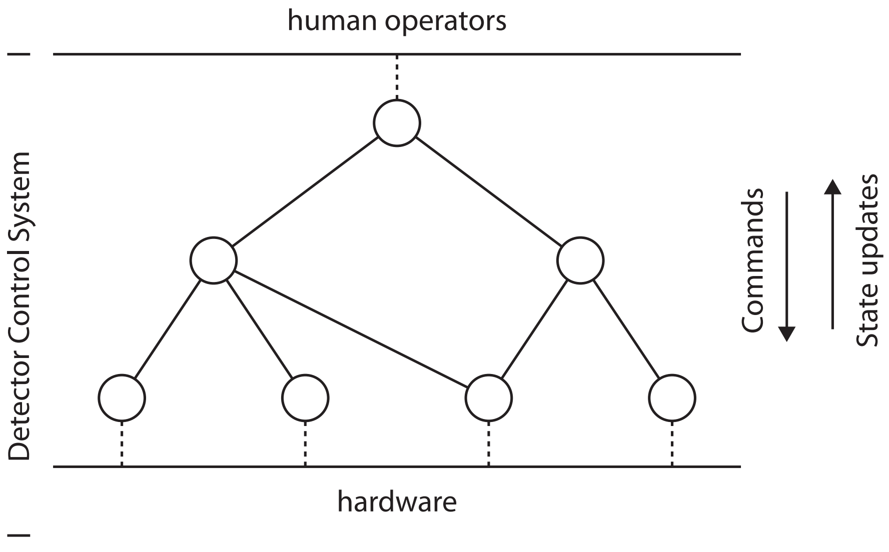


Figure 1.1: Architecture of a Detector Control System

complexity of the system lies in the number of nodes and the communication between them: the DCS of CMS consists of over 32000 nodes and in LHCb's and ATLAS's DCSes there are even more: over 79500 nodes. Their systems yield state spaces of about 10^{25000} and 10^{57600} states. Due to the sheer sizes of the DCSes, getting and maintaining complete overview of a DCS is very hard if not impossible for any human. A complicating factor is that parts of the systems were written and are being maintained by several subgroups of the experiments, probably having different design philosophies. This makes system maintenance and error discovery hard to do by hand. As a result of this, the DCSes sometimes behave differently than developers expect.

Sometimes the DCSes enter a livelock and report different states repeatedly. This causes parts of the DCS to become unresponsive to commands, leaving hardware without controller, possibly resulting in loss of potential physics data and in extreme cases even in damage to expensive scientific equipment. For example: if due to a livelock in the DCS it reports that it is off, no data will be recorded, even though everything is working fine and the detector is in fact ready for detection. Unexpected behaviour might have the same consequences, potentially ruining difficult scientific experiments.

The goal of this project is to help developers of DCSes to improve their software.

Before the start of this project, FSM and CFSM had been formalised by providing an automated translation of the systems to the *micro Common Representation Language 2* (mCRL2) [GMR+09] process algebra. This translation is described in [HWK+11], [HKK+11], and [HKW12]. Using this translation, several desirable properties were identified that can be automatically checked using the mCRL2 toolset[5] [GKM+08]. Due to the large state spaces, verify-

[5]`http://www.mCRL2.org`

ing those properties on complete systems is infeasible. Therefore, dedicated lightweight detection tools were developed for two of the desirable properties: local loops and pairwise reachability [HKK$^+$11]. These *verification tools* ran only on Linux and were not ready for developer use.

We contributed to the project in two ways: a) we improved the existing verification tools and made them available to developers and b) we identified new desirable properties. Moreover we extended and improved the mCRL2 translation in close collaboration with others. A summary of these changes is given in Appendix A.

Both existing verification tools were implemented in the ASF+SDF meta-environment [BvDH$^+$01]. We reimplemented them in Python, added reductions, reduced the number of false-positives and integrated tools into the development environment. We identified several static semantic issues the development environment does not detect and integrated checks for them in all verification tools. The results of this are already visible: the number of issues in the DCS of CMS dropped from 52 to 1. In addition to this, we identified a new kind of livelock: non-local loops. We reasoned Bounded Model Checking (BMC) [BCC$^+$03] could be applied to detect these loops, but detecting them in the DCSes would be infeasible due to the state space sizes. We developed several reductions for state space size, and identified a special case of the new non-local loops. For this special case, we again developed state space reductions and developed a detection tool. This detection tool actually found issues in CMS, ATLAS and ALICE.

Outline The systems and language the nodes are written in are described in Section 2. In Section 3, the static semantic issues we identified are described. The first desirable property, pairwise reachability of states, and how we extended it is described in Section 4. Section 5 shows the second desirable property, local loops, and how we extended it. In Section 6 we describe the potentially unwanted behaviour we identified: non-local loops. We summarise our results and conclude in Section 7.

2 Preliminaries

In this section, we first give an overview of system structures and node behaviour, after that we give the semantics and syntax of the FSM language. We finish with a description of the environment.

2.1 CFSM

Structure A *system* is the part of the software of a DCS that we analysed and consists of *nodes*, that can have parents and children. At the start of the project, we assumed the system had a tree structure, but later we discovered that this assumption was too strict: the 'tree' has multiple roots and some nodes have multiple parents. No nodes were found that are a direct or indirect parent of itself, so the system is modelled as a directed acyclic graph. To make a clear distinction, the nodes without parents are called *sources* in this report.

To illustrate size and structure of a system, Figure 2.1 gives an overview of the detector control system structure of CMS as of May 2012. Appendix B shows the system structures of LHCb, ATLAS and ALICE. These images were generated using software written by Sjoerd Cranen of the Eindhoven University of Technology. In these images, a coloured dot represents a node and a black line represents a parent-child relation. Colours represent node models.

A node without children is a *leaf*. Leaves usually provide an interface to hardware: they gather sensor data and decide to move to a state based on sensor-specific information. Moreover, leaves pass commands to hardware. Leaves are partly modelled: only the states they can be in are modelled, not their other behaviour. To include all possible behaviour, we assume they may change to any state at any moment.

We assume all nodes are independent, except for the messages sent between parents and children. In reality, this is not always the case: a lower level language can be used to express behaviour like 'move to state S if at least 60% of my siblings are in state Y'. As this behaviour cannot be expressed in FSM, we do not model it, i.e., we assume only parent-child relationships exist.

Communication Nodes communicate using messages. A node can send *state update* messages to its parents and *command* messages to its children. The sources have no parents but can receive commands from other external control systems or from human operators. In this report, these external entities are referred to as *grandparents*. As leaves have no children, they cannot receive state updates.

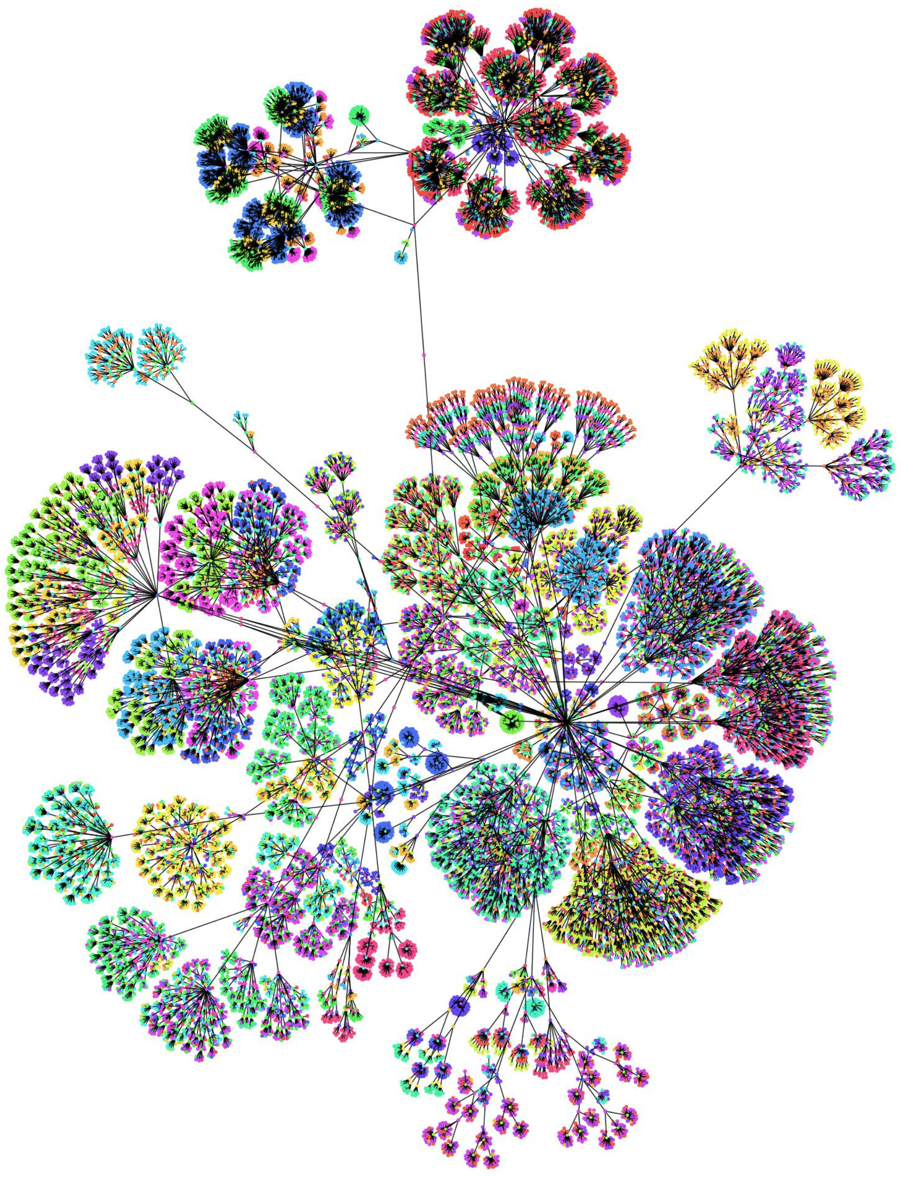


Figure 2.1: Detector control system structure of CMS (image generated using software written by Sjoerd Cranen of the Eindhoven University of Technology)

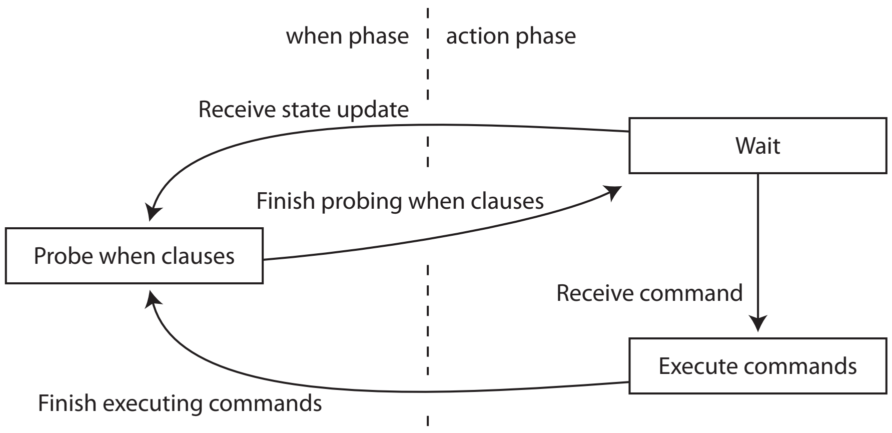


Figure 2.2: Phases of a CFSM node

Messages that cannot be processed immediately are queued. The queues have a maximal capacity and if this is reached, new messages are dropped.

A node sends a state update in four situations: a) at initialization of the system, b) if it changed state, c) if it received a command, or d) if it received a state update.

Control flow In a node, WHEN clauses describe how the node reacts to the states of its children and ACTION clauses describe how the node reacts to the receipt of commands. Figure 2.2 gives a schematic view of the control flow of a node. At every point in time, a node can be in either the WHEN phase or the ACTION phase. In the WHEN phase, the node probes its WHEN clauses and acts according to the first one that matches. If no WHEN clause matches, there is nothing to do for the node and it moves to the ACTION phase. In the ACTION phase, the node waits for commands and if one arrives, it is executed. If all commands in the queue have been executed, the node sends a state update message to its parents and moves to the WHEN phase.

2.2 FSM

In this section, we describe the language in which the nodes are described: FSM. An EBNF grammar of FSM and a description of our parser are given in Appendix C. The syntax and semantics of FSM that are relevant for our analyses are as follows:

CFSM class Every node is a finite state machine which can change state according to the rules in its CFSM class. A *CFSM class* is a model of node behaviour. Its concept is comparable to classes in object-oriented programming: a CFSM class can be instantiated in several nodes. A CFSM class consists of a non-empty list of states, the first being the initial state. Listing 1 contains an example of a CFSM class: the CFSM class is called RPC and consists of two states: ON and ERROR. State ON has a WHEN clause and an ACTION clause.

The WHEN clause makes sure that if any child is in state ERROR, the node will move to state ERROR as well. If the node is in state ON and receives a RESET command, it will execute the ACTION clause by sending an OFF command to all of its children.

```
class: $FWPART_$TOP$RPC
    state: ON
        when ( $ANY$FwCHILDREN in_state ERROR ) move_to ERROR
        action: RESET
            do OFF $ALL$FwCHILDREN
    state: ERROR
        when ( $ANY$FwCHILDREN in_state ON ) move_to ON
```

Listing 1: Example of a CFSM class

States A node is always in exactly one state. A *state* in FSM describes the behaviour of a node when it is in that state, and consists of a, possibly empty, list of WHEN clauses and a, possibly empty, list of ACTION clauses.

when clauses A WHEN *clause* is a guarded instruction. WHEN clauses are probed in the order in which they are given until one is found for which the guard is true. The instruction of this clause, the *referrer*, is executed. The possible referrers are MOVE_TO, DO and STAY_IN_STATE:

- MOVE_TO S makes the CFSM continue execution in state S.
- DO C executes action clause C.
- STAY_IN_STATE S, where S is a state, does nothing, except for preventing further evaluation and execution of WHEN clauses by moving to the ACTION phase, regardless of the value of S (S is optional).

action clauses An ACTION *clause* is a non-empty list of instructions, called *statements*. When the ACTION clause is executed, the statements are executed in the order they are provided. An ACTION clause C is executed when a parent sends a C command, or when the node itself executes a DO C referrer. The possible statements are MOVE_TO, DO, SLEEP, WAIT and IF:

- MOVE_TO S makes the node continue execution in state S.
- DO P CO sends command CO to all children matching child pattern P.
- SLEEP X halts execution for X seconds.
- WAIT P halts execution until all children matching child pattern P have finished processing a previously sent command.
- IF G THEN S1 ELSE S2 ENDIF waits until all children mentioned in G have finished processing a previously sent command. After that either S1 or S2 is executed, depending on whether G holds. The ELSE S2 part is optional.

Guards and child patterns A *guard* is a Boolean statement on the states of children. A basic guard can have two shapes: a) P IN_STATE S, where P is a child pattern and S is either a state or a set of states, or b) P EMPTY, where P is a child pattern. A *child pattern* is a combination of a selector (ANY or ALL) and the name of a CFSM class or the literal `FwCHILDREN`. Assuming the child pattern targets children (there are nodes of the given CFSM class), the semantics of a) are straightforward:

```
$ANY$FwCHILDREN in_state ERROR
```

is true if and only if any child is in state ERROR, and

```
$ALL$RPC_HV in_state {READY, NOT_READY}
```

is true if and only if each child of CFSM class RPC_HV is either in state READY or NOT_READY. The semantics of b) are straightforward as well: the guard

```
$RPC_HV empty
```

is true if and only if the node has no children of CFSM class RPC_HV.

More complex guards can be built using basic guards, brackets and the Boolean operators AND, OR and NOT. Moreover, subclassing can be used: the CFSM class CLASS__&SUBCLASS can be targeted in a guard by using either CLASS or CLASS__&SUBCLASS as child pattern.

Guards use a three-valued logic instead of the common empty domain rules, which disables (parts of) guards that do not refer to any children [Fra11]. A basic `in_state` guard of which the child pattern matches no children evaluates to GHOST. The Boolean operators evaluate as follows:

$$\begin{array}{ll} x \text{ AND GHOST} & = x \\ x \text{ OR GHOST} & = x \\ \text{GHOST AND } x & = x \\ \text{GHOST OR } x & = x \\ \text{NOT GHOST} & = \text{GHOST} \end{array}$$

If an entire guard evaluates to GHOST, it is treated as evaluating to false. For instance, if a node has children, but not of CFSM class RPC_HV, the guard

```
$ALL$RPC_HV in_state {READY, NOT_READY} and
    $ANY$FwCHILDREN in_state ERROR
```

is equivalent to

```
$ANY$FwCHILDREN in_state ERROR
```

Even though these rules are inconsistent with common Boolean reasoning rules, see Appendix D for an example, the implementation works like this and any formalization has to deal with it. Problems are avoided in practice by first removing all GHOST values before applying any other Boolean rules.

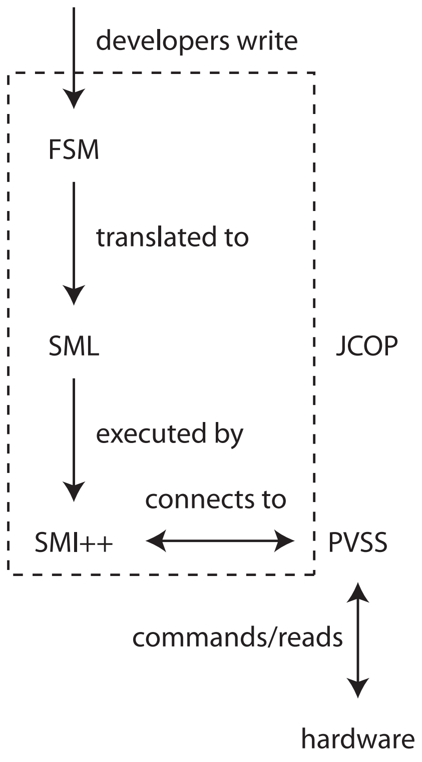


Figure 2.3: Environment overview

2.3 Environment

The development, simulation, and production environments use Siemens' SCADA software, called PVSS. During this project, version 3.8 service pack 2 of PVSS was used, in combination with the JCOP framework version 4.3.0 [HGBGS05], containing the SMI++ framework and FSM-ConfDB version 3.4.4. When this report mentions the *development environment*, it refers to the combination of these programs/plugins, used for developing CFSMs and modifying the system structure. Figure 2.3 shows a schematic overview. The dashed box represents a part of the JCOP framework, written at CERN and used by all experiments.

System structure and node models are stored in the development environment. Developers write CFSM classes that are compiled to executable code automatically. Periodically, the system structure and node models are dumped to a database for backup purposes. The development environment is slow in retrieving all information, and the information is stored in an unstructured manner. Therefore, we retrieve the information from a backup database to a local directory and apply verification tools to this local copy. Each node model is stored in its own file and the parent-child relations are stored in a Comma-Separated Values (CSV) file, denoting the system structure. A second time the system is retrieved from a database, only changed node models are saved to enable caching based on filesystem timestamps.

3 Static semantics

The development environment performs basic syntax checking when a CFSM class is translated to production or simulation code. However, it does not check for static semantic issues. In production systems, static semantic issues are handled at runtime using fall-back mechanisms, causing possibly unexpected behaviour. We identified two classes of static semantic issues: *error*, if the semantics are unclear or ambiguous, and *warning*, if the semantics are clear but the consequences are probably not as intended. Verification tools may crash or, even worse, produce incorrect results when a CFSM class contains these errors. Therefore, we defined and implemented static semantic checking for CFSM classes.

Errors and warnings We identified the following possible static semantic issues:

- A MOVE_TO referrer or statement mentions a state that is not declared;
  This is an error, as there is no definition what should happen once this construct gets executed. When encountered in a running production system, the production environment will detect the error and move the node to an unmodelled DEAD state. A human operator then needs to restart the node manually. A newer version of the development environment is said to catch this error more elegantly, but this version was not available at time of writing.

- A DO referrer mentions an action that is not declared;
  This is an error, as there is no definition of what should happen once this construct gets executed. The development environment will do nothing if this error occurs at runtime and continues as if a STAY_IN_STATE referrer was executed.

- A STAY_IN_STATE referrer mentions a state different from the state it is in;
  This is an error, as it is ambiguous whether the node must stay in its current state or move to the given state. The development environment allows it and stays in the current state.

- A MOVE_TO referrer mentions the state it is in;
  This is a warning, since its meaning is well defined, but it is likely the developer meant to use a STAY_IN_STATE referrer. On execution, the development environment will remain in the same state and in the WHEN phase.

- A CFSM class is declared more than once within a file;
  This is an error, as it is ambiguous as which of the two CFSM classes a node should behave. It cannot occur in practice as every CFSM class is stored in a separate file. We found no occurences of this error, but add it for sake of completeness and future use.

- A state is declared more than once within a CFSM class;
  This is an error as it is ambiguous to which state to go when a MOVE_TO construct pointing to the common state name is executed. We did not find a way to enter two states having the same name within a class in the development environment, but add this error for sake of completeness and future use.

- An action is declared more than once within a state.
  This is an error as it is ambiguous which action to execute when it is called. We did not find a way to enter two actions having the same name within a state in the development environment, but add this error for sake of completeness and future use.

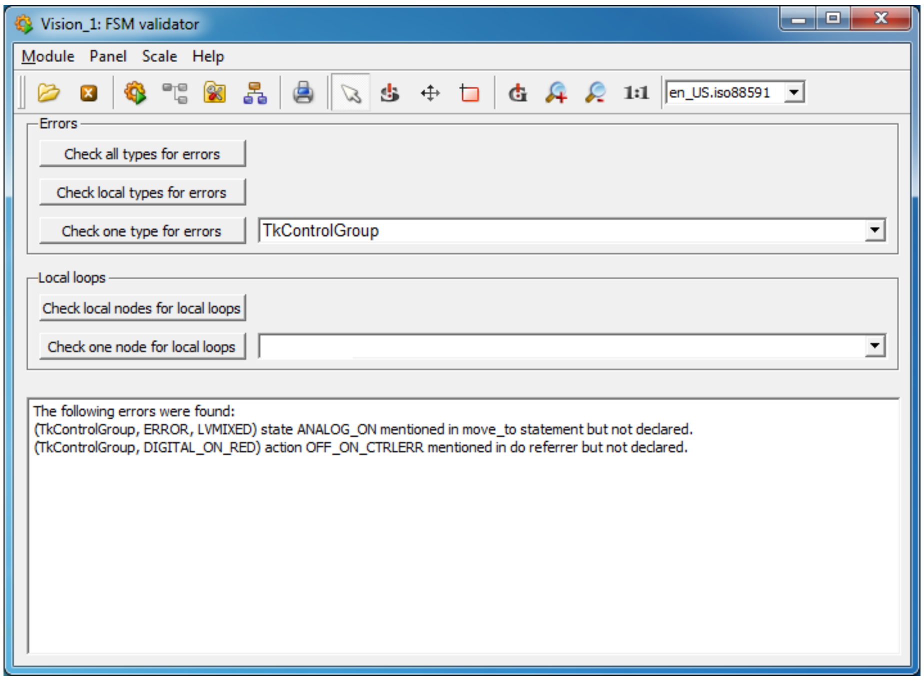
The 'A MOVE_TO statement mentions the state it is in' is considered neither an error nor a warning, as its semantics are equivalent to letting the statement block finish normally and there is no reason to assume the developer meant something else.

3.1 Tool

Checks for these static semantic issues were integrated in our parser, so that every verification tool automatically uses it. Please refer to Appendix C for more details regarding the Python parser. Static semantic errors are treated in the same way as parser errors: if the parser finds a static semantic error, it is reported to the user and the tool exits after all CFSM classes have been checked. The parser will report warnings, but will not exit if one was found. Hence, every verification tool generates a syntax and static semantic issues report.

PVSS In order to assist developers in detecting and avoiding static semantics issues, a checking ability was integrated in the development environment PVSS. Figure 3.1 shows a screenshot of the user interface (*panel* in PVSS terms). A developer can either check all CFSM classes in the whole experiment, all CFSM classes on her computer or a specific CFSM class. This integration is available to all DCS developers within CERN and uses separate files in order not to interfere with core functionality.

Web interface integration *CMS Online* is a web environment being developed allowing developers to walk through the system structure, see the statuses of nodes and examine the FSM code. The PVSS integration requires installation before developers can use it and is not very suitable for checking large numbers of nodes and CFSM classes. To avoid this, the checks for CMS were added to the existing *CMS online* web environment. The checks are performed on a snapshot of the CMS system and the results are stored in a database. As


Vision_1: FSM validator
Module Panel Scale Help
en_US.iso88591
Errors
Check all types for errors
Check local types for errors
Check one type for errors
TkControlGroup
Local loops
Check local nodes for local loops
Check one node for local loops

```
The following errors were found:
(TkControlGroup, ERROR, LVMIXED) state ANALOG_ON mentioned in move_to statement but not declared.
(TkControlGroup, DIGITAL_ON_RED) action OFF_ON_CTRLERR mentioned in do referrer but not declared.
```

Figure 3.1: PVSS panel showing result of static semantic check

can be seen in the screenshot in Figure 3.2, developers can filter the results by type, here errors or warnings, and by subdetector. We implemented the checks and storage of the results, the web interface was implemented by a developer in close collaboration with us.

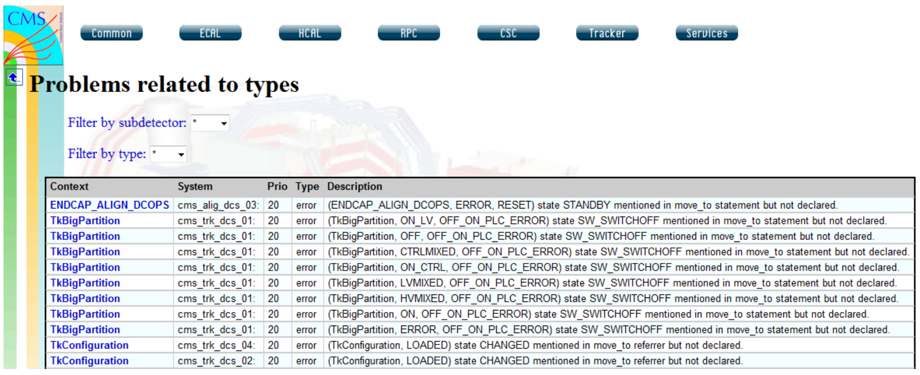
3.2 Results

We performed static semantic checks on all experiments, see Table 3.1 for the results of these checks. As this project originates from CMS, we obtained its system four times.

The static semantic checks have proven useful: when an engineer saw the reported errors, he expressed the wish to have been able to use these checks back in 2006 (when most CFSMs were developed). The data in Table 3.1 show that the static semantic issues present in the DCS of CMS dropped drastically during the project (from 52 to 1). To illustrate the errors and the reports the verification tools generate, Table 3.2 shows three errors that were present in CMS as of November 2011 and were confirmed to be fixed in January 2012. Engineers confirmed this was a direct result of this project.

Other tools might be unable to work on CFSM classes containing errors. In order for us to be able to use the verification tools on real data, and not having to wait for developers to fix the errors, the errors reported were manually corrected in our local copy in agreement with the developers. For instance, the error

```
state ANALOG_ON mentioned in move_to referrer but not declared.
```



Problems related to types

Filter by subdetector: *

Filter by type: *

Context	System	Prio	Type	Description
ENDCAP_ALIGN_DCOPS	cms_alig_dcs_03:	20	error	(ENDCAP_ALIGN_DCOPS, ERROR, RESET) state STANDBY mentioned in move_to statement but not declared.
TkBigPartition	cms_trk_dcs_01:	20	error	(TkBigPartition, ON_LV, OFF_ON_PLC_ERROR) state SW_SWITCHOFF mentioned in move_to statement but not declared.
TkBigPartition	cms_trk_dcs_01:	20	error	(TkBigPartition, OFF, OFF_ON_PLC_ERROR) state SW_SWITCHOFF mentioned in move_to statement but not declared.
TkBigPartition	cms_trk_dcs_01:	20	error	(TkBigPartition, CTRLMIXED, OFF_ON_PLC_ERROR) state SW_SWITCHOFF mentioned in move_to statement but not declared.
TkBigPartition	cms_trk_dcs_01:	20	error	(TkBigPartition, ON_CTRL, OFF_ON_PLC_ERROR) state SW_SWITCHOFF mentioned in move_to statement but not declared.
TkBigPartition	cms_trk_dcs_01:	20	error	(TkBigPartition, LVMIXED, OFF_ON_PLC_ERROR) state SW_SWITCHOFF mentioned in move_to statement but not declared.
TkBigPartition	cms_trk_dcs_01:	20	error	(TkBigPartition, HVMIXED, OFF_ON_PLC_ERROR) state SW_SWITCHOFF mentioned in move_to statement but not declared.
TkBigPartition	cms_trk_dcs_01:	20	error	(TkBigPartition, ON, OFF_ON_PLC_ERROR) state SW_SWITCHOFF mentioned in move_to statement but not declared.
TkBigPartition	cms_trk_dcs_01:	20	error	(TkBigPartition, ERROR, OFF_ON_PLC_ERROR) state SW_SWITCHOFF mentioned in move_to statement but not declared.
TkConfiguration	cms_trk_dcs_04:	20	error	(TkConfiguration, LOADED) state CHANGED mentioned in move_to referrer but not declared.
TkConfiguration	cms_trk_dcs_02:	20	error	(TkConfiguration, LOADED) state CHANGED mentioned in move_to referrer but not declared.

Figure 3.2: Static semantic issues shown in CMS Online

		errors	in CFSM classes	warnings	in CFSM classes
CMS	11-2011	52	22 of 571 (4%)	1	1 of 571
	01-2012	48	19 of 571 (3%)	1	1 of 571
	03-2012	43	17 of 566 (3%)	0	0 of 566
	05-2012	1	1 of 571 (0.2%)	0	0 of 571
LHCb	11-2011	68	30 of 1028 (3%)	6	6 of 1028
ATLAS	05-2012	19	7 of 1098 (0.6%)	3	3 of 1098
ALICE	05-2012	43	16 of 579 (3%)	5	5 of 579

Table 3.1: Results of static semantic checks

```
Error found in .\ECALfw_Dee(38909).fsm.
 (ECALfw_Dee, OFF_LOCKED) action NEUTRALISE mentioned in
 do referrer but not declared.

Error found in .\ECALfw_Supermodule(38738).fsm.
 (ECALfw_Supermodule, OFF_LOCKED) action NEUTRALISE mentioned in
 do referrer but not declared.

Error found in .\CMSfwLhcHandshakeCU(38562).fsm.
 (CMSfwLhcHandshakeCU, ADJUST_WARNING) action NOTFIY_STANDBY
 mentioned in do referrer but not declared.
```

Table 3.2: Three static semantic errors that were fixed in CMS

was fixed by adding an empty state ANALOG_ON to the CFSM class.

For ALICE, this method was not sufficient. A small number of CFSM classes contains low-level SML code that our parser and tools cannot handle. In order to be able to perform the verification checks on ALICE, we removed this SML code by hand. Based on manual inspection, we assume this did not significantly change behaviour.

In the remainder of this report, we assume all CFSM classes are free of static semantic errors. Appendix F shows the chain of verification tools, described later on in this report, that we performed after the static semantic checks.

4 Pairwise reachability

A desirable property of nodes is that all states in it should remain reachable at all times during execution. Two states X and Y are *pairwise reachable* if and only if it is possible to start in X, move to Y and move back to X again. It does not matter how this is achieved: commands from parents, going through other states, and state changes from children may be required to do so. If not all states of a node are pairwise reachable, it is possible to get trapped in a subset of states.

Pairwise unreachability can indicate disabled security mechanisms and nodes unable to leave a state. However, pairwise unreachability does not imply error. For instance, a CFSM class can be reused in a node with a different set of children, where some states are not in use intentionally. In this section, we describe the verification tool that existed before start of this project, we introduce a combinations reduction to shorten computation time, introduce a state space reduction for later use and conclude with a description of our improved tool and its results.

Previous work Pairwise reachability as a desirable property for CFSM is described in [HKW12] and [HKK$^+$11]. Before the start of this project, a tool checking for pairwise reachability was present, implemented in ASF+SDF. This tool, given a combination of a parent and its children, computes pairwise reachability and outputs the result in textual and graphical format. The tool lacked support for several FSM constructs, for example the IF statement, and ran only on Linux.

4.1 Checks

We first introduce some notation: a *configuration* c is a partial function that maps nodes to states, denoting for each node the state in which it currently is. For readability reasons, we use a tuple notation. For instance, a configuration denoting node 1 and 3 being in state A and node 2 being in state B is denoted as:

```
(1: A, 2: B, 3: A)
```

In this report, $c[n \to s]$ denotes the function c in which element n maps to s.

Checking Pairwise reachability of states can be approximated using knowledge of only the combination of a parent and its direct children: a *parent-children* *combination*. To calculate pairwise reachability for a parent-children combination t, we transform the problem to a graph problem. A directed graph is constructed containing all states of the parent n. An edge is added from state X to state Y if and only if there is a configuration c of t such that n can move from X directly to Y. In the resulting graph, strongly connected components are computed. If there are multiple strongly connected components, by definition not all states are pairwise reachable (X and Y are pairwise reachable if and only if they are in the same strongly connected component).

Checking whether it is possible to move from state X to state Y is done by generating a Boolean formula that is true if and only if it is possible to move from state X to state Y in configuration c for t, assuming any command can be received by n. Using a satisfiability (SAT) solver it is determined whether such a configuration c exists. A quick check whether a MOVE_TO Y construct is even present in state X reduces the number of SAT calls.

Images To enable developers to grasp the potential problems quickly, the tool outputs images. The tool present before the start of this project generated images; we redesigned them a bit. In these images, ellipses are states, rectangles are strongly connected components, the initial state is coloured green and arrows denote state changes that can happen in the node. Arrows that do not cross strongly connected component bounds are coloured grey. Using this colour coding, we highlight problems and hide everything else. Figure 4.1 shows an example.

Further study Our pairwise reachability check assumes a node can receive any command from its parents. In reality, this is not necessarily the case, as a parent can never send a specific command. Hence, the pairwise reachability checks can give false negatives. An interesting field of further study would be to determine which commands could actually be received by a node and to resolve the circular dependencies that arise from it. (It would also imply that knowledge of parents is required, making the checks global instead of local)

4.2 Minimising combinations

Computing pairwise reachability for all pairs of states in all parent-children combinations of a system is a lengthy computation. Therefore, we first introduce a partial order relation between parent-children combinations and second, we use the partial order relation to reduce the number of parent-children combinations that needs to be checked.

4.2.1 Partial order

Say t is a parent-children combination of a parent and its direct children. Define (p, f) to be a tuple representing t, where p is the CFSM class of the parent in t and f is a function from CFSM classes to integers, denoting for each CFSM class how many children in t are of that CFSM class. Using this, we define a partial order relation $\leqslant$ on parent-children combinations:

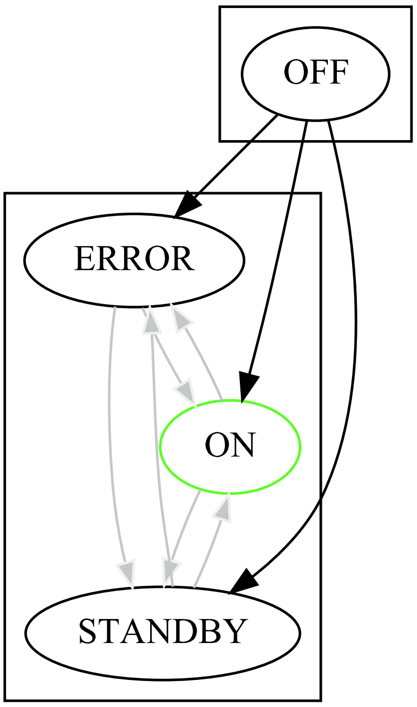


Figure 4.1: Output of pairwise reachability check, showing that state OFF is not reachable

Definition 4.2.1 *Given two parent-children combinations $t_1 = (p_1, f_1)$ and $t_2 = (p_2, f_2)$, define $t_1 \leqslant t_2$ to be true if and only if all of the following hold:*

- *The parent nodes are of the same CFSM class: $p_1 = p_2$;*
- *The frequencies of all CFSM classes in t_1 are all smaller than or equal to the corresponding frequencies in t_2, but the sets of used CFSM classes are the same: for all c it holds that $f_1(c) \leq f_2(c)$ and $f_1(c) = 0 \Leftrightarrow f_2(c) = 0$.*

Note that if $t_1 \leqslant t_2$ one can transform t_1 into t_2 by adding child nodes until all frequencies match: $\mathit{duplicate}((p, f), c) = (p, f[c \to f(c) + 1])$, where (p, f) is a parent-children combination and c is a CFSM class, assuming $f(c) \neq 0$.

Example Observe two parent-children combinations x and y: x is a combination of a node of class Q with 2 children of class A and y is a combination of a node of class Q with 3 children of class A. Let us denote them by $x = (p_x, f_x) = (\text{Q}, (\lambda c.0)[\text{A} \to 2])$ and $y = (p_y, f_y) = (\text{Q}, (\lambda c.0)[\text{A} \to 3])$. Then $x \leqslant y$ since the class of both parents is Q, in both combinations only A is used for the children and $f_x(\text{A}) \leq f_y(\text{A})$.

Using Definition 4.2.1, we prove that $\leqslant$ is a partial order relation:

Lemma 4.2.2 *$\leqslant$ is a partial order relation*

Proof

- Reflexivity: $(p_1, f_1) \leqslant (p_1, f_1)$
  It holds that $p_1 = p_1$ and for all c it holds that $f_1(c) \leq f_1(c)$ and $f_1(c) = 0 \Leftrightarrow f_1(c) = 0$, so $\leqslant$ is reflexive.

- Anti-symmetry: if $(p_1, f_1) \leqslant (p_2, f_2)$ and $(p_2, f_2) \leqslant (p_1, f_1)$, then $(p_1, f_1) = (p_2, f_2)$.
  Assuming $(p_1, f_1) \leqslant (p_2, f_2)$ and $(p_2, f_2) \leqslant (p_1, f_1)$ then by the first clause of the definition $p_1 = p_2$, and for all c it holds that $f_1(c) \leq f_2(c)$ and $f_2(c) \leq f_1(c)$. By anti-symmetry of $\leq$ on integers, $\leqslant$ is anti-symmetric.

- Transitivity: if $(p_1, f_1) \leqslant (p_2, f_2)$ and $(p_2, f_2) \leqslant (p_3, f_3)$ then $(p_1, f_1) \leqslant (p_3, f_3)$.
  Assuming $(p_1, f_1) \leqslant (p_2, f_2)$ and $(p_2, f_2) \leqslant (p_3, f_3)$ then $p_1 = p_3$ by transitivity of equivalence on CFSM classes.

$$(\forall c : f_1(c) \leq f_2(c) \wedge (f_1(c) = 0 \Leftrightarrow f_2(c) = 0)) \wedge$$
$$(\forall c' : f_2(c') \leq f_3(c') \wedge (f_2(c') = 0 \Leftrightarrow f_3(c') = 0))$$
$$\Rightarrow \{\text{dummy transferring}\}$$
$$\forall c : f_1(c) \leq f_2(c) \leq f_3(c) \wedge (f_1(c) = 0 \Leftrightarrow f_2(c) = 0) \wedge$$
$$(f_2(c) = 0 \Leftrightarrow f_3(c) = 0)$$

  By transitivity of $\leq$ and $\Leftrightarrow$, $\leqslant$ is transitive.

As $\leqslant$ is reflexive, anti-symmetric and transitive, $\leqslant$ is a partial order relation. ∎

4.2.2 Combinations reduction

Using the partial order $\leqslant$, the number of parent-children combinations that needs to be checked can be reduced. To prove this, we use a rephrased version of [HKK+11, Lemma 2]:

Lemma 4.2.3 *Given two children of the same CFSM class that are both in the same state, removing one of these children will not affect any decision that a parent takes in the* WHEN *phase.*

We first prove that the reachability graph of t_1 is a subgraph of the reachability graph of t_2 if $t_1 \leqslant t_2$.

Lemma 4.2.4 *Suppose two parent-children combinations t_1 and t_2 such that $t_1 \leqslant t_2$. Then the reachability graph of t_1 is a subgraph of the reachability graph of t_2.*

Proof The reachability graph of t_1 contains an edge from state A to state B if and only if there is a configuration of the children in t_1 such that a MOVE_TO B statement or referrer in state A can be executed. Say n is a child of t_1. Add one child n', of the same CFSM class as n, to t_1 to obtain t_1': $t_1' = \mathit{duplicate}(t_1, n)$.

Consider an edge (A, B) in the reachability graph of t_1. Then there is a configuration c of children in t_1 such that a MOVE_TO B statement or referrer is executed in state A of the parent. Choose c' to be a configuration in t_1' such that it is a copy of c, with the addition of n', which is in the same state as n: $c' = c[n' \to c(n)]$.

By Lemma 4.2.3, no guard of the parent can distinguish between c and c'. Hence, there is a configuration in t_1' in which the MOVE_TO B statement or referrer is executed in state A of the parent and edge (A, B) is present in the reachability graph of t_1' as well. As no edge can disappear by adding nodes of which the CFSM class was already present, the reachability graph of t_1 is a subgraph of the reachability graph of t_1'. More general, the reachability graph before applying *duplicate* is a subgraph of the reachability graph after applying *duplicate*. As t_2 can be obtained from t_1 by applying *duplicate* repeatedly and by transitivity of $\leqslant$, the reachability graph of t_1 is a subgraph of the reachability graph of t_2. ∎

If we have a t_1 and t_2 such that $t_1 \leqslant t_2$, then by Lemma 4.2.4 the reachability graph of t_1 is a subgraph of the reachability graph of t_2. Hence, the graph of t_2 can be obtained from the graph of t_1 by adding edges. By monotonicity of strongly connected components, adding edges to a graph can only cause merging of strongly connected components. If t_1 does not contain multiple strongly connected components, t_2 does not contain multiple strongly connected components either. If t_1 contains strongly connected components, t_2 can only contain the strongly connected components of t_1, but some might be merged.

By default we only check t_1, the smallest combination, to save computation time. For instance, given the two parent-children combinations
$(\textsc{q}, (\lambda c.0)[\textsc{a} \to 2])$ and $(\textsc{q}, (\lambda c.0)[\textsc{a} \to 3])$, only $(\textsc{q}, (\lambda c.0)[\textsc{a} \to 2])$ would be checked. To prevent engineers deciding not to fix the multiple connected components based on information about t_1 only, reports mention both t_1 and t_2. Users of the tool can choose not to apply this reduction, by checking thoroughly. Checking *thoroughly* only reduces two parent-children combinations t_1 and t_2 if they are equal up to renaming.

4.3 Reachability reduction

With an under-approximation of the states that are unreachable in a node, a state space reduction is possible. Consider a node n with an initial state s_i and another state s_u. If s_u is unreachable from s_i, s_u can be removed from n without altering the possible behaviour of n. Obviously, s_i itself is always reachable. As no MOVE_TO s_u construct is ever executed, they are replaced with a random statement: MOVE_TO s_i. This last step is necessary to prevent introducing a static semantic error. The complete procedure *reachability reduction* is shown in Listing 2.

Reachable states are computer on parent-children combinations. The CFSM class of the parent might be in use in other parent-children combinations. To distinguish the reduced and the original CFSM class, our implementation makes a copy of the CFSM class and gives it a new unique identifier.

```
function REACHABILITYREDUCTION($t$)
    $reachableStates \leftarrow reachabilityCheck(t)$
    $str \leftarrow$ string representation of $t.parent.CFSMClass$
    $initialState \leftarrow t.parent.CFSMClass.states[0]$
    for $state \in t.CFSMClass.states$ do
        if $i \notin reachableStates$ then
            replace "move_to $state$ " with "move_to $initialState$ " in $str$
            remove state $state$ from $str$
        end if
    end for
    $t.parent.CFSMClass \leftarrow parse(str)$
end function
```

Listing 2: Reachability reduction, where t is a parent-children combination, *reachabilityCheck*() is a function taking a parent-children combination t and returning the reachable states of the parent of t, and *parse* is our FSM parser

4.3.1 Restrictions

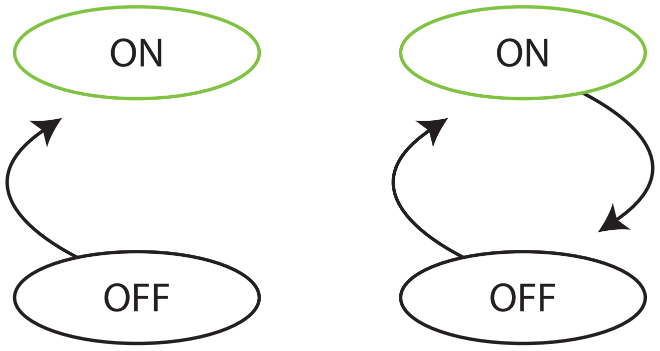
One should take care not to apply reachability reduction before a procedure that alters the system structure, as reachability of states might change: observe the WHEN clause with guard in Listing 3. Assume that the MOVE_TO Y is the only MOVE_TO Y construct in the CFSM class, that Y is not the initial state and that CFSM class CLASS2 has a reachable state X. The guard is never satisfiable when there is a child of CFSM class CLASS1 present. Therefore, reachability reduction will remove state Y. After that, suppose a reduction R removes all children of CLASS1. Then by the used three-valued logic, the first two conjuncts of the guard are ignored, the guard becomes satisfiable, Y becomes reachable and hence should not have been removed.

Hence, reachability reduction should not be applied before any test or reduction that changes the system structure without further study of the consequences.

```
when (
  $ALL$CLASS1 in_state X and
  $ALL$CLASS1 not_in_state X and
  $ANY$CLASS2 in_state X )
    move_to Y
```

Listing 3: A possibly unsatisfiable guard in a WHEN clause

Another restriction is that the combinations reduction described in Section 4.2 cannot be used before reachability reduction: consider two parent-children combinations t_1 and t_2, such that $t_1 \leqslant t_2$. Then by Lemma 4.2.4 the reachability graph of t_2 is a subgraph of the reachability graph of t_1. To illustrate this, Figure 4.2 shows two possible reachability graphs of t_1 and t_2. If the combinations reduction as described is applied before reachability reduction, states are removed in t_1 and t_2 based on the reachability graph of t_1. In our example in Figure 4.2, state OFF is removed. In t_2 however, state OFF is reachable and should not be removed. Hence, combinations reduction as described cannot



(a) Reachability graph of parent-children combination t_1

(b) Reachability graph of parent-children combination t_2

Figure 4.2: Reachability graphs

be used before reachability reduction. However, as the reachability graph of t_1 is a subgraph of the reachability graph of t_2, we can safely remove states based on the reachable states of t_2.

4.4 Tool

The existing pairwise reachability tool was written in the ASF+SDF meta-environment, which only runs on Linux, while most developers of CFSM use Windows. Therefore, we had to replace the ASF+SDF meta-environment and we chose Python in combination with Picoparse. For more details about this see Appendix E. We reimplemented the existing verification tool in Python and made it work on both Linux and Windows. Moreover, we extended the tool to accept a complete system and to support all FSM constructs. The tool checks all parent-children combinations of the system, using multiple threads to speed up the process.

Generating reports and reducing the system cannot use the same combinations reduction. Therefore, the user must choose between either only generating a report, only reducing the system or do both but thoroughly.

To limit the number of reports, equal pairwise-reachability problems are grouped in post-processing. Two reports are considered equal if and only if all of the following hold:

- The parents that contain the problems are of the same CFSM class;
- Both problems contain the same sets of strongly connected components;
- In both problems, the same connections between strongly connected components exist.

We added an option to generate an HTML report, which can be seen in Figure 4.3. It shows that the parent (RPC_DEVICE) cannot return to the OFF

```
Not all states of this parent are always pairwise reachable:
parent RPC_Device (39134)

The parent can walk through its states as described in the image,
while starting in the green state. In some combinations, there
are states where you cannot make a roundtrip to.

This might happen in the following nodes:
combination                                       parent nodes
1*RPC_Device (39134), 1*RPC_Boards (39641),       RPC_ENDCAP_HWUSC
1*RPC_BC (39647), 1*RPC_MAO (39649),              (cms_rpc_dcs_06:,
1*RPC_SY1527 (39651)                               120264)
```

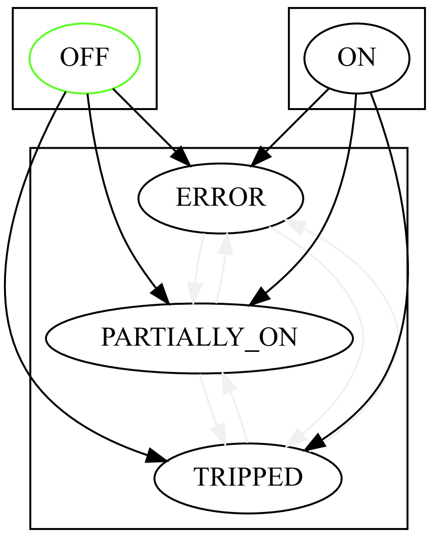
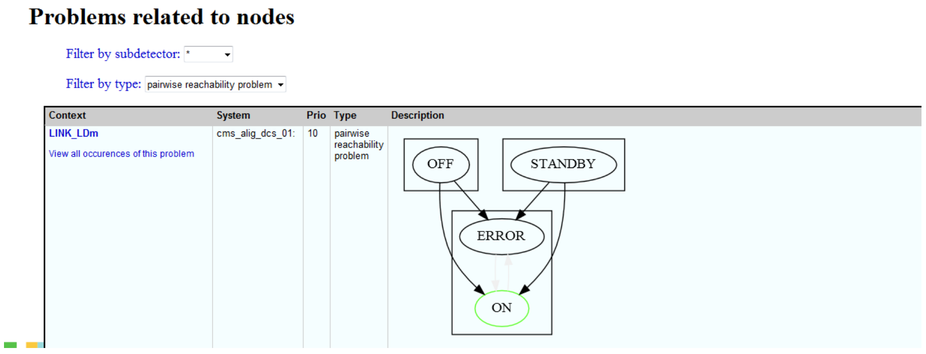


Figure 4.3: Pairwise reachability report


Problems related to nodes

Filter by subdetector: *

Filter by type: pairwise reachability problem

Context	System	Prio	Type	Description
LINK_LDm View all occurences of this problem	cms_alig_dcs_01:	10	pairwise reachability problem	

Figure 4.4: Pairwise reachability results shown in CMS Online

state once it has left it. It can also never reach the ON state. The report also shows that this can happen in only one node: RPC_ENDCAP_HWUSC.

4.4.1 Development environment integration

In order to give developers access to pairwise reachability checks, results were added to CMS Online. The developers considered the images self-explanatory, so the explanations from the HTML reports were left out. See Figure 4.4 for a screenshot of the web integration of pairwise reachability. Users can filter the results to show only potential problems of nodes within their own part of the detector. The combinations reduction potentially produces false positives in exchange for computation time. As the computation time is in the order of minutes for all experiments, combinations reduction was disabled: all checks are thorough.

We want the web interface application to accept systems that contain errors: if it does not, useful results would be hidden from developers if only one of the CFSM classes used contains a static semantic error. Pairwise reachability detection checks parent-children combinations, so any parent-children combination in which no erroneous CFSM classes are used can be checked safely. We do not want to change the pairwise reachability tool itself, so we come up with a pre-processing step: consider a parent p, having two children n and n', such that n is of an erroneous CFSM class. If n is removed, the behaviour of p might change. Therefore, the tool disconnects all children from p, making it a leaf. The same is done for n: all its children are disconnected from it. Finally, n is removed. If any of the disconnected nodes has no parents left, it is made a source. This procedure, shown in Figure 4.5, is applied for all erroneous nodes n, such that the system contains no erroneous nodes and we can use the pairwise reachability checks without any modifications.

4.5 Results

Combinations reduction results We performed the pairwise reachability checks twice: once with the combinations reduction using the partial order $\leqslant$, and once thoroughly. Table 4.1 shows the results of the combinations re-

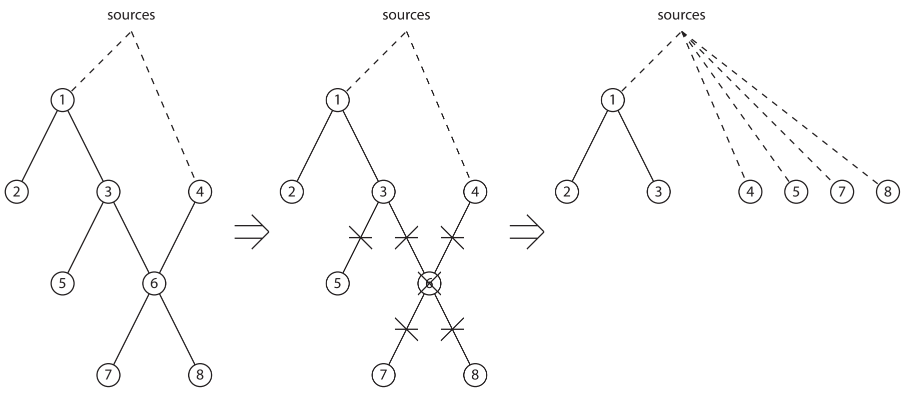


Figure 4.5: Removal of erroneous nodes: node 6 cannot be parsed and is removed. A circle with a number is a node, a line is a parent-child relation, where the topmost node is the parent. Dashed lines mean that the node is included in the set. Left: situation before, center: node 6 is removed, right: situation after.

		Combinations checked		Nodes reported	
		thoroughly	- reduced	thoroughly	- reduced
CMS	11-2011	568	- 402	927	- 927
	01-2012	568	- 401	1353	- 1353
	03-2012	564	- 398	1313	- 1313
	05-2012	578	- 409	903	- 903
LHCb	11-2011	1106	- 883	430	- 430
ATLAS	05-2012	1746	- 985	2089	- 2096
ALICE	05-2012	410	- 356	1836	- 1836

Table 4.1: Results of combinations reduction before pairwise reachability (reports in ATLAS: thoroughly: 187, reduced: 184)

duction. Applying combinations reduction only altered the output in ATLAS, where seven (2089 - 2096) false positive nodes were reported as a result of the reduction and three reports were incorporated in other reports. Due to the significant reduction of combinations checked and the minimal impact on results, we suggest applying the combinations reduction.

Check results Pairwise reachability check results, with combinations reduction, are shown in Table 4.2. As the behaviour of leaves is not modelled and therefore unknown, the number of nodes in the results is taken not counting leaves.

All of the tested experiments contain nodes with multiple strongly connected components. It is remarkable that the number of reports (62 to 75 (+17%)) and nodes with possible strongly connected components (927 to 1353 (+30%)) increased from November 2011 to January 2012 in CMS. A possible explanation is that developers had no access to reachability reports during these months and during this time there was a technical stop, meaning that all experiments at the LHC were stopped and that hardware upgrades, software updates and system

		reports	nodes with pairwise unreachable states		
CMS	11-2011	62	927	of 9038	(10%)
	01-2012	75	1353	of 9038	(15%)
	03-2012	74	1313	of 9045	(15%)
	05-2012	50	903	of 9064	(10%)
LHCb	11-2011	78	430	of 16139	(2.7%)
ATLAS	05-2012	184	2096	of 12963	(16%)
ALICE	05-2012	95	1836	of 4623	(40%)

Table 4.2: Results of pairwise reachability checks

		states removed	rounds	state space before - after
CMS	11-2011	451	5	10^{26438} - 10^{26400}
	01-2012	485	5	10^{26227} - 10^{26186}
	03-2012	469	5	10^{26315} - 10^{26276}
	05-2012	397	5	10^{26402} - 10^{26368}
LHCb	11-2011	410	6	10^{58363} - 10^{58325}
ATLAS	05-2012	3125	11	10^{59209} - 10^{58922}
ALICE	05-2012	2310	6	10^{12158} - 10^{12052}

Table 4.3: Results of reachability reduction

changes were carried out.

Reachability reduction results We applied reachability reduction with combinations reduction repeatedly, until the reduction stabilised. After that, we applied reachability reduction thoroughly until that stabilised. Table 4.3 shows the results of these reductions. We compute the *state space* by taking the product of the number of states in all nodes (including leaves). In no experiment except ATLAS, the thorough reachability reduction removed states after reachability reduction with combinations reduction stabilised. In ATLAS, thorough reachability reduction removed seven extra states in two rounds. We conclude that even though a reduction factor of 10^{106} (ALICE, 10^{12158} to 10^{12052}) is huge, we conclude that impact on state space sizes is small.

5 Local loops

A desirable property of nodes is that they are free of livelocks. A *local loop* is a kind of livelock in which a node traverses through some of its states, while its children do not change state. In each state, the node executes WHEN clauses and moves to a next state. While doing this, a node will remain in the WHEN phase. Therefore, it will not process any commands and is out of control. Unless the children change their state such that the local loop ends, the only way for operators to end a local loop is to reboot the node, or manually order its subsystems to change state. Moreover, looping behaviour can propagate to higher level nodes, so that in the control room operators observe flickering of several lights. Engineers and operators sometimes jokingly call this a Christmas tree.

In this section, we first give an example of a local loop. After that, we describe the work that had been done before start of this project, introduce a parent-children combinations reduction, describe how our tools work and give the results of it.

Example Figure 5.1 shows an example of a local loop: node 1 is of the CFSM class given in the figure and has two children: node 2 and node 3. Suppose this combination reaches the following configuration:

```
(1: ON, 2: ERROR, 3: ON)
```

Then the WHEN clause of state ON is enabled, so node 1 moves to state ERROR:

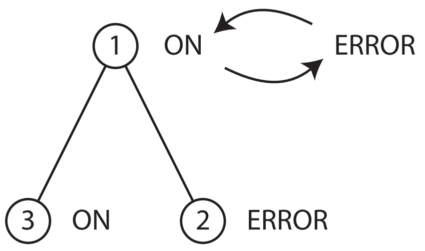
```
(1: ERROR, 2: ERROR, 3: ON)
```

After that, the first WHEN clause of state ERROR is enabled, so node 1 moves to ON again, completing the loop:

```
(1: ON, 2: ERROR, 3: ON)
```

Previous work Local loops were described in [HKW12], [HKK$^+$11], and [Kus11]. A *local loop* being present in a parent-children combination means that given some configuration, the parent keeps changing state while the children do not change state. Usually, a local loop is a chain of WHEN clauses having MOVE_TO referrers that point to each other in a circular manner.

A tool that checks a system for local loops, implemented in ASF+SDF, was already present before the start of this project. It used a theoretical upper bound on the number of children per CFSM class to avoid checking of all parent-children combinations. In [HKK$^+$11, Section 5.1] and [Kus11] it is



Where node 1 consists of the following FSM code:

```
state: ON
    when ( $ANY$FwCHILDREN in_state ERROR ) move_to ERROR
state: ERROR
    when ( $ANY$FwCHILDREN in_state ON ) move_to ON
```

Figure 5.1: Example of a local loop: node 1 loops between states ON and ERROR

proven that if a certain artificial parent-children combination is local loop free, all parent-children combinations (covered by the artificial combination) are local loop free as well. This reduces the number of combinations that have to be checked, but the downside is that some reported local loops are false positives. For instance, we found a local loop that could only occur in a parent-children combination having at least two children of class X. However, there were no parent-children combinations having two children of class X in the system and hence the loop could not occur. This might, as stated in [HKK+11], ultimately result in developers ignoring the output of the tool.

Therefore, the tool written during this project only checks combinations actually present in the system. This reduces the number of false positives being reported, but requires a recheck if the system is changed whenever the number of children changes. We believe that this last problem is solved by the running time of our tools: our tool usually takes less than a minute to check CMS.

In the local loops described in [HKW12], during the loops commands can be sent. During this project, a developer encountered an example in which such a command intentionally caused a child to change state, preventing the local loop from happening. To avoid reporting these false positives, we consider loops in which a command is sent non-local. Non-local loops are described in Section 6.

Checking To detect local loops, a parent-children combination t is translated to a Boolean formula that is true if and only if the children of t are in a fixed configuration and the parent of t can start in a state, run and end up in the start state again. This formula is satisfiable if and only if there is local loop in the parent-children combination. The formula is checked using a SAT solver. If the formula is satisfiable, the solver provides evidence consisting of a list of configurations. Using these configurations, it is determined which states of the parent are traversed during the loop and which WHEN clause is enabled in each state. Using this information, developers confirmed they can easily trace and understand the local loop.

Restrictions Local loop detection as we described it finds at most one local loop in each parent-children combination. However, several local loops might be present. There are ways to detect all local loops in a parent-children combination. One of them is to use a satisfiability solver that provides all satisfying assignments to a formula, another one is to adapt the formula whenever a local loop is found, by adding clauses excluding the local loops found. This procedure is repeated as long as new local loops are found. We suggest implementation of this latter procedure.

5.1 Minimising combinations

In order to minimise the number of parent-children combinations that need to be checked for local loops, the partial order relation $\leqslant$, as described in Section 4.2.1, can be used. We first prove that given two parent-children combinations t_1 and t_2, such that $t_1 \leqslant t_2$ and t_2 is local loop free, then t_1 is local loop free as well. Second, we introduce a reduction based on this. This reduction is similar to the reduction mentioned in [HKW12], but guarantees that if a local loop is reported, it is present in at least one node in the system.

Theorem 5.1.1 *Assume two parent-children combinations t_1 and t_2 such that $t_1 \leqslant t_2$. If t_2 contains no local loop, then t_1 contains no local loop either.*

Proof Suppose that $t_1 = (p_1, f_1)$ contains a local loop. Then there is a configuration c, such that the parent p_1 can loop through some of its states. Say n is a child in t_1. Add one child n', of the same CFSM class as n, to t_1 to obtain t'_1: $t'_1 = \mathit{duplicate}(t_1, n)$. Construct a new configuration c' on t'_1 by copying c and adding child n' in the same state as child n: $c' = c[n' \to c(n)]$.

By Lemma 4.2.3 no guard in p_1 can distinguish between configurations c and c' and hence t'_1 contains a local loop as well. As applying *duplicate* preserves local loops and t_2 is derivable from t_1 by applying *duplicate* repeatedly, t_2 contains a local loop as well. By contraposition, if t_2 contains no local loop, t_1 cannot contain a local loop either. ∎

The choice was made to only check t_2 and, in case of a local loop, assume that the local loop is present in both t_1 and t_2. This saves computing time and potentially reports. Moreover, if an engineer fixes t_2, t_1 is fixed as well. The downside is that an engineer might decide not to fix the local loop based on information about t_2 only, leaving a possible local loop in t_1 unnoticed. To address this, both t_1 and t_2 are mentioned in the report as potentially containing a local loop. The developer can also choose to skip this optimization by applying the checks *thoroughly*. When checking thoroughly, only equal parent-children combinations are checked together.

Future work The artificial parent-children combinations reduction proven in [Kus11] is an upper bound: only a limited number of children that are of a certain CFSM class needs to be considered to prove local loop presence or absence. Applying this upper bound to existing parent-children combinations might reduce the number and complexity of parent-children combinations to be checked. We suggest further study and implementation.

5.2 Tool

Motivated by the same reasons as described in Section 4.4, we reimplemented the local loop tool in Python. It collects all parent-children combinations, selects the parent-children combination to check using combinations reduction and checks them for local loops. To speed up the process, it uses multiple threads to execute this last step.

This procedure produces seemingly duplicate reports. To limit the number of reports, equal loops are grouped in post-processing. Two loops are considered equal if and only if all of the following hold:

- The parents that contain the loops are of the same CFSM class;
- Both loops run through the same state names in the same order;
- In both loops, in the same state, the same when clauses are enabled.

These requirements ensure that developers do not get duplicate reports, while the grouping ensures still all information is shown. The information shown in the report was determined in close cooperation with developers. Figure 5.2 shows an example of a report: it gives the CFSM class of the parent (`CmsBrmCuType`), the states the local loop runs through (`ERROR` and `STANDBY`), the configuration of the children that enable the local loop, the WHEN clauses that are enabled and all nodes in which this local loop can occur (`CMS_BRM`).

5.2.1 Development environment integration

PVSS integration We added local loop checking to the same panel as the static semantic issues checker, so that developers can access it easily. On this panel, developers can either check a single node for local loops or check all nodes on her computer. Needless to say, this integration checks and reports for static semantic errors and warnings first. A screenshot of the PVSS panel is shown in Figure 5.3.

Web interface integration Local loop checks have been added to the web integration in CMS Online, that was described in Section 4.4.1. The combinations reduction described saves time but potentially produces false positives. As a database dump takes in the order of hours and the combinations reductions usually saves less than a minute, this reduction was disabled by checking thoroughly. Before applying local loop detection, erroneous nodes are removed and the system is reduced using reachability reduction, both are described in Section 3.1. Figure 5.4 is a screenshot of the web integration of local loops.

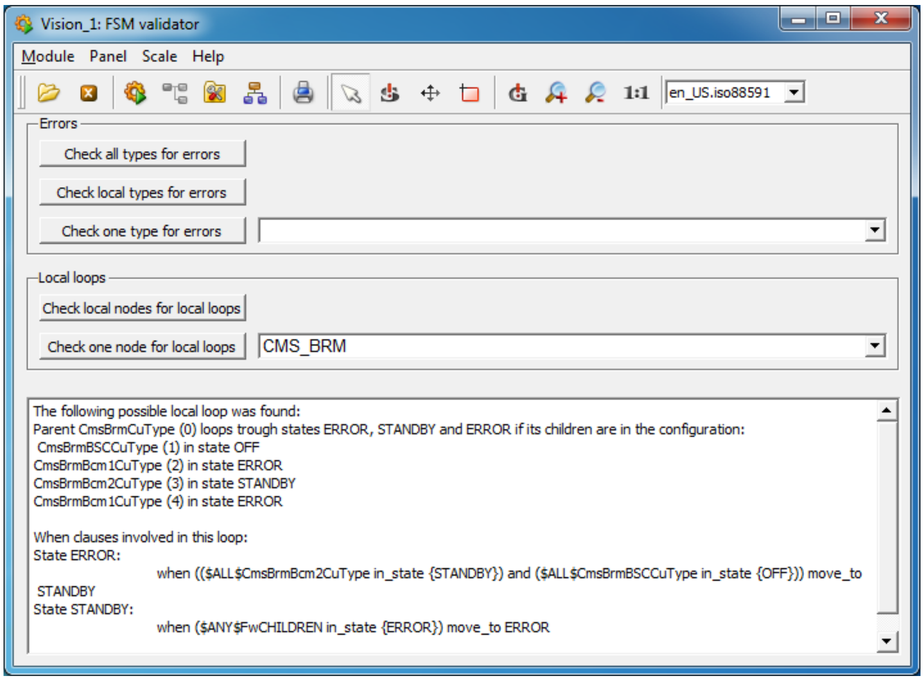
```
This parent contains a local loop:
CmsBrmCuType (39133, cms_brm_dcs_01:)

The parent can walk through states ERROR, STANDBY and ERROR in
some parent-children combinations. For instance, the parent will
loop if it has the following children in states:
CmsBrmBcm1CuType(39192) in state ERROR
CmsBrmBSCCuType(39516)  in state OFF
CmsBrmBcm1CuType(39192) in state ERROR
CmsBrmBcm2CuType(39468) in state STANDBY


When clauses involved in this loop:
  state: ERROR
    when (($ALL$CmsBrmBcm2CuType in_state {STANDBY})
      and ($ALL$CmsBrmBSCCuType in_state {OFF})) move_to STANDBY
  state: STANDBY
    when ($ANY$FwCHILDREN in_state {ERROR}) move_to ERROR


This might happen in the following parent-children combinations:
combination                    parent nodes
1*CmsBrmBSCCuType(39516)       120266: CMS_BRM
2*CmsBrmBcm1CuType(39192)
1*CmsBrmBcm2CuType(39468)
```

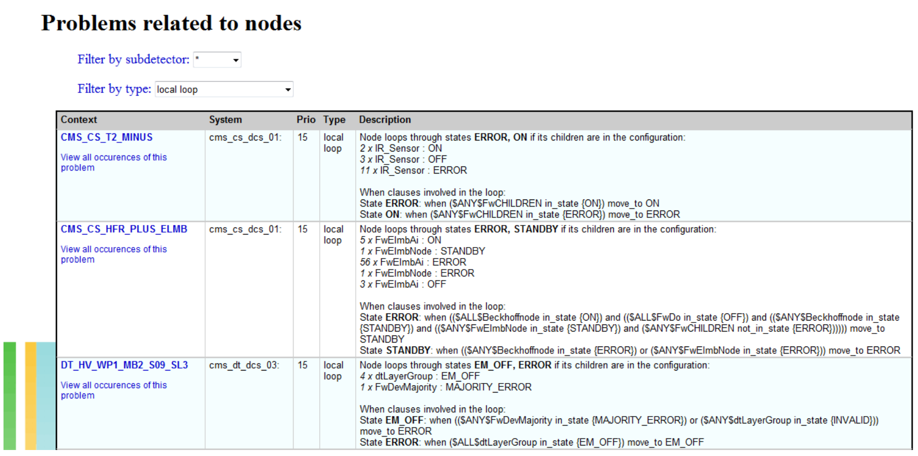
Figure 5.2: Local loop report


Vision_1: FSM validator
Module
Panel
Scale
Help
en_US.iso88591
Errors
Check all types for errors
Check local types for errors
Check one type for errors
Local loops
Check local nodes for local loops
Check one node for local loops
CMS_BRM

```
The following possible local loop was found:
Parent CmsBrmCuType (0) loops trough states ERROR, STANDBY and ERROR if its children are in the configuration:
 CmsBrmBSCCuType (1) in state OFF
CmsBrmBcm1CuType (2) in state ERROR
CmsBrmBcm2CuType (3) in state STANDBY
CmsBrmBcm1CuType (4) in state ERROR

When clauses involved in this loop:
State ERROR:
             when (($ALL$CmsBrmBcm2CuType in_state {STANDBY}) and ($ALL$CmsBrmBSCCuType in_state {OFF})) move_to
 STANDBY
State STANDBY:
             when ($ANY$FwCHILDREN in_state {ERROR}) move_to ERROR
```

Figure 5.3: PVSS integration: panel with result of local loop detection



Problems related to nodes

Filter by subdetector: *

Filter by type: local loop

Context	System	Prio	Type	Description
CMS_CS_T2_MINUS View all occurences of this problem	cms_cs_dcs_01:	15	local loop	Node loops through states **ERROR, ON** if its children are in the configuration: *2 x* IR_Sensor : ON *3 x* IR_Sensor : OFF *11 x* IR_Sensor : ERROR When clauses involved in the loop: State **ERROR**: when (ANYFwCHILDREN in_state {ON}) move_to ON State **ON**: when (ANYFwCHILDREN in_state {ERROR}) move_to ERROR
CMS_CS_HFR_PLUS_ELMB View all occurences of this problem	cms_cs_dcs_01:	15	local loop	Node loops through states **ERROR, STANDBY** if its children are in the configuration: *5 x* FwElmbAi : ON *1 x* FwElmbNode : STANDBY *56 x* FwElmbAi : ERROR *1 x* FwElmbNode : ERROR *3 x* FwElmbAi : OFF When clauses involved in the loop: State **ERROR**: when ((ALLBeckhoffnode in_state {ON}) and ((ALLFwDo in_state {OFF}) and ((ANYBeckhoffnode in_state {STANDBY}) and ((ANYFwElmbNode in_state {STANDBY}) and (ANYFwCHILDREN not_in_state {ERROR}))))) move_to STANDBY State **STANDBY**: when ((ANYBeckhoffnode in_state {ERROR}) or (ANYFwElmbNode in_state {ERROR})) move_to ERROR
DT_HV_WP1_MB2_S09_SL3 View all occurences of this problem	cms_dt_dcs_03:	15	local loop	Node loops through states **EM_OFF, ERROR** if its children are in the configuration: *4 x* dtLayerGroup : EM_OFF *1 x* FwDevMajority : MAJORITY_ERROR When clauses involved in the loop: State **EM_OFF**: when ((ANYFwDevMajority in_state {MAJORITY_ERROR}) or (ANYdtLayerGroup in_state {INVALID})) move_to ERROR State **ERROR**: when (ALLdtLayerGroup in_state {EM_OFF}) move_to EM_OFF

Figure 5.4: Local loop check results in CMS Online

		Combinations checked		Nodes with local loops	
		thoroughly	- reduced	thoroughly	- reduced
CMS	11-2011	568	- 406	1695	- 1698
	01-2012	458	- 405	1659	- 1662
	03-2012	564	- 402	1546	- 1547
	05-2012	578	- 413	1302	- 1303
LHCb	11-2011	1113	- 899	33	- 33
ATLAS	05-2012	1757	- 1009	107	- 107
ALICE	05-2012	414	- 361	701	- 701

Table 5.1: Results of combinations reduction before local loop detection

		reports	nodes with local loops		
CMS	11-2011	24	1698	of 9038	(19%)
	01-2012	24	1662	of 9038	(18%)
	03-2012	22	1547	of 9045	(17%)
	05-2012	19	1303	of 9064	(14%)
LHCb	11-2011	8	33	of 16139	(0.2%)
ATLAS	05-2012	8	107	of 12963	(0.8%)
ALICE	05-2012	45	701	of 4623	(15%)

Table 5.2: Results of local loop checks

5.3 Results

We applied the local loop verification tool to the systems of all four large LHC experiments. We first describe the results of combinations reduction, after which we describe the results of the local loop checks themselves. We finish the results with a local loop that happened in practice.

Combinations reduction results Table 5.1 shows the results of combinations reduction. The number of reports did not change in any of the tested experiments, probably because the loops would be equal anyway and filtered out in post-processing. We observe that the combinations reduction reduces the number of combinations to be checked in all experiments by between 12 and 43%. The reduction causes no change in reports and introduces at most three false positives (CMS 11-2011: 1695 to 1698 nodes). Therefore, we applied combinations reduction before the local loop detection.

Check results As described in [HKK$^+$11], local loops can be proven present or absent by the tool very quickly. In at most a few minutes (uncached: 28s sec, cached: 19 sec for CMS May 2012), the necessary parent-children combinations are checked and the reports are generated. Table 5.2 gives an overview of local loop search results. Starting point was the systems as reduced by reachability reduction. We can safely use reachability reduction before local loop detection, as reachability reduction only removes states that are unreachable. It implies however that we will not find local loops in states that are removed by reachability reduction.

```
Wed Feb 22 12:07:05 2012 - [CMS_BRM] in state [STANDBY]
Wed Feb 22 12:07:05 2012 - [CMS_BRM] in state [ERROR]
Wed Feb 22 12:07:05 2012 - [CMS_BRM] in state [STANDBY]
Wed Feb 22 12:07:05 2012 - [CMS_BRM] in state [ERROR]
...
Wed Feb 22 12:07:05 2012 - [CMS_BRM] in state [STANDBY]
Wed Feb 22 12:07:05 2012 - [CMS_BRM] in state [ERROR]
Wed Feb 22 12:07:05 2012 - [CMS_BRM] in state [STANDBY]
Wed Feb 22 12:07:05 2012 - [CMS_BRM] in state [ERROR]
```

Figure 5.5: Log file of a local loop in CMS

Local loops in production A manual search of the log files of CMS revealed that some of the reported local loops had actually occurred in its production system. For instance, the local loop of which Figure 5.2 shows the report, happened in February 2012 in CMS. The log file belonging to this problem is shown in Figure 5.5. It shows that the node CMS_BRM looped between states ERROR and STANDBY, exactly as predicted by our tool. The log file as we gathered it was incomplete, but shows that the node changed state at least 38 times per second. The local loop was confirmed to be fixed in March 2012. Moreover, another local loop was confirmed to be fixed, that was present in 112 nodes.

The local loop described in Appendix G happened in at least three nodes in November 2011. An inspection of the CFSM class involved resulted in the discovery of a copy-paste error. As of March 2012, the node in which it happened does not contain a local loop anymore.

LHCb confirmed that a reported local loop had happened in their DCS, the report saved time needed to search for the cause of the loop. Unfortunately, we did not obtain a log of this local loop. The discovery of these reported local loops happening in practice increased confidence in and support for this project.

As no systematic log search tools are available, searching for occurrences of loops in log files has to be done in a web environment while making use of search terms and intuition. It is very time consuming. Developing tools to automate this task might be an interesting field of further study.

6 Non-local loops

Local loops are livelocks within a single node. In this section, we introduce another kind of loop, that involves multiple nodes: non-local loops. We first give an example, an informal definition and explanation why non-local loops are undesirable. We give a formal definition of non-local loops in Section 6.1. In Section 6.2 we argue that checking real systems for non-local loops is infeasible due to state spaces sizes. To reduce these state spaces, we describe reductions in Section 6.3. In Section 6.4 we describe a subclass of non-local loops, state-keeping non-local loops, for which we developed a check that is feasible on CMS, ATLAS and ALICE.

A *non-local loop* is a serialised infinite trace of messages, that contains every message sent between nodes of a system s, so that the sources of s receive no commands and the leaves of s do not change state infinitely often without infinitely often receiving a command.

Annotated configurations Before we give an example, we extend the concept of configurations with message queues to obtain annotated configurations. An *annotated configuration* is a configuration extended with a message queue for each node. For instance, the annotated configuration

```
(1: (A, []), 2: (B, [Y]), 3: (A, [Y]))
```

denotes the configuration `(1: A, 2: B, 3: A)`, extended with an empty command queue for node 1 and command queues containing one Y message for both node 2 and 3. The commands in the queue have been received by the node, but have not yet been executed.

For the sake of feasibility of the analyses, we abstract from message queuing: for non-local loops we assume that each node can queue one command message and that state update messages are handled immediately. We expect this abstraction to exclude a limited amount of behaviour. For instance: suppose that a node is able to send a command Y, but only after sending command X. As all nodes are independent, we are allowed to postpone sending of command Y until command X is processed, avoiding the queue size limitation. We presume that by postponing sending of commands, only a minimal amount of behaviour is excluded.

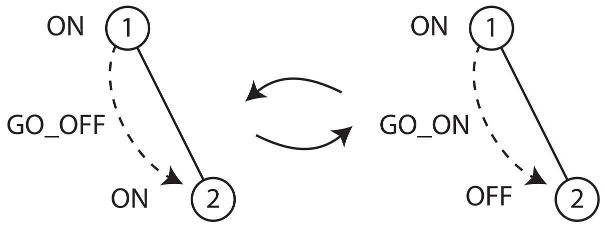


Figure 6.1: Two times system s with a non-local loop: node 1 keeps sending GO_ON and GO_OFF commands

Example Observe the system described in Figure 6.1: a system s consisting of node 1 of class PARENT, given in Listing 4, having one child: node 2. Node 2 is of class CHILD2, that is given in Listing 5. When node 2 is in state ON, node 1 sends a GO_OFF command. Upon receipt of this command, node 2 moves to state OFF. When node 2 is in state OFF, node 1 sends a GO_ON command, upon which node 2 moves to state ON. Obviously, this can be repeated infinitely often.

```
class: $FWPART_$TOP$parent
    state: ON
        when ( $ANY$CHILD2 in_state ON ) do SWITCH_OFF
        when ( $ANY$CHILD2 in_state OFF ) do SWITCH_ON
        action: SWITCH_ON
            do GO_ON $ALL$CHILD2
        action: SWITCH_OFF
            do GO_OFF $ALL$CHILD2
```

Listing 4: Class PARENT

```
class: $FWPART_$TOP$CHILD2
    state: ON
        action: GO_OFF
            move_to OFF
    state: OFF
        action: GO_ON
            move_to ON
```

Listing 5: Class CHILD2

Before we formalise this non-local loop, we describe the loop in more detail: suppose that s starts in annotated configuration

```
(1: (ON, []), 2: (ON, []))
```

As the first WHEN clause of node 1 is enabled, node 1 executes action SWITCH_OFF by sending a GO_OFF command to node 2:

```
(1: (ON, []), 2: (ON, [GO_OFF]))
```

Node 2 processes this command and moves to state OFF:

```
(1: (ON, []), 2: (OFF, []))
```

Then, the second WHEN clause of node 1 is enabled, so node 1 executes action SWITCH_ON by sending a GO_ON command to node 2:

```
(1: (ON, []), 2: (OFF, [GO_ON]))
```

Node 2 processes this command and moves to state ON, bringing the system back in the initial annotated configuration:

```
(1: (ON, []), 2: (ON, []))
```

6.1 Formal

Notation Before we give a formal definition of non-local loops, we introduce some formal notation to describe the system structure. Say s is a system and n is a node.

$n \in s$	denotes that n is a node (not a grandparent) of s;
$S(s)$	is the set of sources of s;
$L(s)$	is the set of leaves of s;
$P(n)$	is the set of parents of n. In case $n \in S(s)$, $P(n)$ is the set of grandparents of s;
$CH(n)$	is the set of children of n;
$(\mathrm{co}, n, n', com)$	denotes node or grandparent n sending a command message to node n' giving command com.
$(\mathrm{st}, n, n', sta)$	denotes node n sending a state update message to node or grandparent n' announcing node n is in state sta.

Using these notations, we express some requirements on messages, where m is a trace of messages sent in a system s:

- A node can only receive commands from its parents or, in case the node is a source, the system's grandparents:

$$\forall_{n,n',com}\,((\mathrm{co}, n', n, com) \in m \Rightarrow n' \in P(n))$$

- A node can only receive state updates from its children:

$$\forall_{n,n',sta}\,((\mathrm{st}, n', n, sta) \in m \Rightarrow n' \in CH(n))$$

Using these notations, our non-local loop example produces the following message trace (we added the annotated configurations again for clarity):

```
(1: (ON, []), 2: (ON, []))
    (co, 1, 2, GO_OFF)
(1: (ON, []), 2: (ON, [GO_OFF]))
    (st, 2, 1, OFF)
(1: (ON, []), 2: (OFF, []))
```

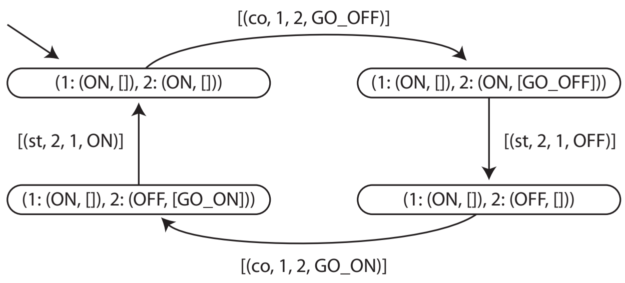

[(co, 1, 2, GO_OFF)]
(1: (ON, []), 2: (ON, []))
(1: (ON, []), 2: (ON, [GO_OFF]))
[(st, 2, 1, ON)]
[(st, 2, 1, OFF)]
(1: (ON, []), 2: (OFF, [GO_ON]))
(1: (ON, []), 2: (OFF, []))
[(co, 1, 2, GO_ON)]

Figure 6.2: Part of a labelled transition system, denoting the non-local loop of our example

```
    (co, 1, 2, GO_ON)
(1: (ON, []), 2: (OFF, [GO_ON]))
    (st, 2, 1, ON)
(1: (ON, []), 2: (ON, []))
```

As the initial and last annotated configuration are equal, node 1 receives no commands and node 2 does not change state infinitely often without infinitely often receiving a command, the trace denotes a non-local loop in s. A non-local loop will flood the system with messages, filling up queues and possibly causing parts of the system to become unresponsive. A non-local loop might also report different states repeatedly: the christmas tree (mentioned in the introduction of Section 5).

Labelled transition system To formalise non-local loops, we model the behaviour of a system s as a labelled transition system. A *labelled transition system* is a four-tuple $(\Sigma, S, s_0, \rightarrow)$, where Σ is a set of labels, S is a set of states, s_0 is the initial state and $\rightarrow$ is a labelled transition relation between states. A labelled transition system starts in state s_0 and uses the relation $\rightarrow$ to move between states in S.

We define a labelled transition system $\mathbb{L}(s)$ for each system s, where S is the set of all possible annotated configurations of s, Σ is the set of all lists of messages and s_0 is the initial annotated configuration of s (i.e. the initial configuration combined with an empty command queue for every node). We define the labelled transition relation such that $ac_0 \xrightarrow{m_0} ac_1$ holds if and only if system s can start in annotated configuration ac_0, run, thereby sending all and only the messages in message list m_0, and end up in annotated configuration ac_1. We define $ac \xrightarrow{[]} ac$ to hold for all annotated configurations ac. Figure 6.2 shows a graphical representation of a part of $\mathbb{L}$ of our example.

Then obviously every non-local loop is a cycle

$$ac_1 \xrightarrow{m_1} ac_2 \xrightarrow{m_2} ac_2 \ldots ac_n \xrightarrow{m_n} ac_1$$

in the labelled transition system $\mathbb{L}$. We summarise this path to a tuple (ac_1, m), where ac_1 is an annotated configuration on the circular path and m is the list of messages sent on the cycle, starting in ac_1: $m = m_1 \mathbin{+\!\!+} m_2 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} m_n$.

An annotated configuration ac is *valid* for a system s if it is a complete and correct function. I.e. $ac(n)$ is defined for all nodes $n \in s$ and if $ac(n)$ denotes a state sta for a node n, the CFSM class of n has state sta. A trace m is *valid* and *complete* for a system s and a configuration ac_1 if $\mathbb{L}(s)$ has a path

$$ac_1 \xrightarrow{m_1} ac_2 \xrightarrow{m_2} ac_3 \ldots ac_{n-1} \xrightarrow{m_n} ac_n$$

for some $ac_2 \ldots ac_n$ such that $m = m_1 +\!\!+\ m_2 +\!\!+ \cdots +\!\!+ m_n$, i.e., such that s can run and send all messages in m (valid), not sending anything else (complete), while starting in ac_1.

Connection with mCRL2 These labelled transition systems are a manual abstraction from the mCRL2 translation of CFSM, that is briefly described in Appendix A. In the mCLR2 translation a system s is described using a labelled transition system. To obtain our labelled transition systems, we abstract from all transitions and state parameters except those relevant to non-local loops: states, command queues and messages.

Formal Formally stated, a non-local loop ℓ is represented by a tuple (ac_1, m), where ac_1 is an initial annotated configuration and m is a non-empty finite list of messages, denoted by the tuples introduced earlier.

Definition 6.1.1 *(ac_1, m) is a* non-local loop *for a system s if and only if all of the following hold:*

- *ac_1 is a valid annotated configuration for s and m is a valid and complete trace for s starting in ac_1.*
- *The trace is infinite and indivisible: s is back in the initial annotated configuration ac_1 if and only if all messages in m have been sent:*

$$ac_1 \xrightarrow{m} ac_1 \wedge$$

$$\forall_{m_1,m_2} \Big((m = m_1 +\!\!+\ m2 \wedge ac_1 \xrightarrow{m_1} ac_1 \xrightarrow{m_2} ac_1) \Rightarrow (m_1 = [] \vee m_2 = [])\Big)$$

- *No source receives a command:*

$$\forall_{n \in S(s)} \neg \exists_{n',com} (co, n', n, com) \in m$$

- *No leaf changes state during the loop without receiving a command:*

$$\forall_{n \in L(s)} ((\exists_{x,x',sta \neq sta'} (st, n, x, sta) \in m \wedge (st, n, x', sta') \in m) \Rightarrow$$

$$(\exists_{n',com} (co, n', n, com) \in m))$$

  As $ac_1 \xrightarrow{m} ac_1$, every node must be in the same state before and after execution of m. Hence if a node changes state in m, it must change back to the initial state in m as well. By current implementation of CFSM, if a node changes state it sends a state update. Hence, we can detect a node changing state during the loop by looking for two state updates sent by that node announcing different states. Moreover, as m is repeated infinitely often, a node that eventually stabilises cannot change state in m at all.

The following properties hold for non-local loops in the current implementation:

- A node n only sends a state update at initialization, if it changed state, it received a command or it received a state update. As m is repeated infinitely often, the initialization is excluded. Therefore, if there is a state update from node n in m, then there must be a state change of n in m, n must receive a command or n must receive a state update:

$$\begin{aligned} &\forall_{s;n \in s}((\exists_{n',sta}(\mathrm{st}, n, n', sta) \in m) \Rightarrow \\ ((&\exists_{n',n'',sta \neq sta'}(\mathrm{st}, n, n', sta) \in m \wedge (\mathrm{st}, n, n'', sta') \in m) \vee \\ &(\exists_{n',com}(\mathrm{co}, n', n, com) \in m) \vee \\ &\exists_{n',sta}(\mathrm{st}, n', n, sta) \in m)) \end{aligned} \tag{6.1}$$

- As a leaf cannot receive state updates, it only sends a state update if it changed state or it received a command:

$$\begin{aligned} &\forall_{s;n \in L(s)}((\exists_{n',sta}(\mathrm{st}, n, n', sta) \in m) \Rightarrow \\ ((&\exists_{n',n'',sta \neq sta'}((\mathrm{st}, n, n', sta) \in m \wedge (\mathrm{st}, n, n'', sta') \in m)) \vee \\ &\exists_{n',com}(\mathrm{co}, n', n, com) \in m) \end{aligned} \tag{6.2}$$

Reachability abstraction We abstract from whether the initial annotated configuration ac_1 is reachable. If the tools report a system to be non-local loop free, it is non-local loop free with and without this abstraction. Moreover, this eases the use of SAT solvers to find non-local loops. Reachability reduction, that was described in Section 4.3, might reduce the possibility of unreachable non-local loops being reported.

Local loops In Section 5, we described local loops. A local loop is not guaranteed to be a non-local loop as well: consider a system as described in Figure 6.3. Suppose that in this system, node 2 has a local loop that is only present if node 3 is in state ON. Node 1 could prevent the local loop from happening by sending a command to node 3 such that node 3 changes its state. Then, the local loop is not a non-local loop, as the trace it would produce cannot happen and hence is not valid.

Using the local loop tool, as described in Section 5.2, local loops can be found quickly. Therefore it is safe to assume all nodes are local loop free without hiding problematic behaviour and it is hence not necessary to include local loops in the analysis of non-local loops:

Assumption 6.1.2 *When studying non-local loops in a system s, assume all nodes of s are local loop free.*

A local loop in a node n means that n changes state infinitely often without receiving or sending a command, or the children of n changing state. Consider an infinite message trace m' in which n contains no local loop, n receives finitely many commands, n sends a finite number of commands and n's children change state finitely often. Then there exists a point p on m' after which n receives no commands, n sends no commands and no child of n changes state. As n is

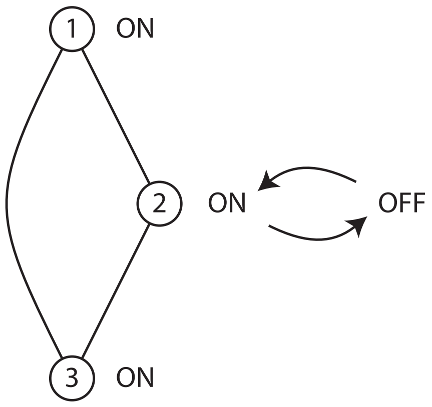


Figure 6.3: A system with a local loop but possibly without a non-local loop: node 2 has a local loop, but node 1 might send a command to node 3 kicking it out of state ON

local loop free, n will change state finitely many times after p in m'. Hence, n will eventually stabilise. This implies that if there is a finite message trace m that is repeated infinitely often and a node n changes state in m, n must either receive a command, send a command, or a child of n must change state in m. Formally stated:

$$\begin{aligned}\forall_n(\exists_{x,x',sta\neq sta'}((\mathrm{st},n,x,sta)\in m\wedge(\mathrm{st},n,x',sta')\in m)\Rightarrow(\\ (\exists_{n',com}(\mathrm{co},n',n,com)\in m)\vee\\ (\exists_{n',com}(\mathrm{co},n,n',com)\in m)\vee\\ \exists_{n'\in CH(n),sta\neq sta'}((\mathrm{st},n',n,sta)\in m\wedge(\mathrm{st},n',n,sta')\in m)))\end{aligned}\tag{6.3}$$

where m is a finite trace that is repeated infinitely often. Using this, we prove that every non-local loop consists of infinitely many command messages and infinitely many state update messages.

Commands Using the formal definition of non-local loops and the assumption that all nodes are free of local loops, we prove that in every non-local loop at least one command is sent.

Lemma 6.1.3 *Every non-local loop $\ell=(ac_1,m)$ contains a command message: $\exists_{n',n'',com}(co,n',n'',com)\in m$.*

Proof We prove $\exists_{n',n'',com}(\mathrm{co},n',n'',com)\in m$ by contradiction: suppose there is a system s having a non-local loop $\ell=(ac_1,m)$ such that m contains no commands:

$$\neg\exists_{n',n'',com}(\mathrm{co},n',n'',com)\in m\tag{6.4}$$

Let k be a list containing all nodes of s. Assume k is sorted in reverse topological order: if node n is a parent of node n', then n' must appear in k before n. As

s is a directed acyclic graph, such an ordering exists. Define $q(i)$ to hold if and only if the node n at position i in k sends no state update:

$$q(i) \equiv \neg\exists_{n',sta}(\mathrm{st}, n, n', sta) \in m$$

We prove by induction on k that no node sends a state update; i.e. $\forall_i q(i)$.

Base case: $q(0)$.
Let n be the node in k at position 0. Observe that n does not have any children and hence is a leaf: $n \in L(s)$.

As $n \in L(s)$, by definition of ℓ, n will eventually stabilise unless it receives a command in m:

$$\begin{aligned}(\exists_{x,x',sta\neq sta'}(\mathrm{st}, n, x, sta) \in m \wedge (\mathrm{st}, n, x', sta') \in m) \Rightarrow \\ \exists_{x,com}(\mathrm{co}, x, n, com) \in m\end{aligned} \tag{6.5}$$

Node n is a leaf and does not change state in m without receipt of a command in m. It was assumed n receives no commands, so n does not change state. Leaves only send a state update on receipt of a command or after a state change. Hence, n sends no state updates in m. Formally stated, by (6.2), (6.5) and (6.4), n sends no state update:

$$\neg\exists_{n',sta}(\mathrm{st}, n, n', sta) \in m$$

Hence, $q(0)$ holds.

Induction step: assume as induction hypothesis that for all j smaller than i, $q(j)$ holds. Let n be the node in k at position $i+1$. Perform a case distinction on whether n is a leaf:

- Case: n is a leaf: $n \in L(s)$:
  Analogous to the proof of the base case: $q(i+1)$.
- Case: n is a node with children: $n \notin L(s)$:
  By the ordering of k, all children of n are in k at positions j with $j \leq i$. By the induction hypothesis, no child of n sends a state update:

  $$\forall_{n' \in CH(n)}\neg\exists_{n'',sta}(\mathrm{st}, n', n'', sta) \in m) \tag{6.6}$$

  n can only receive state updates from its children:

  $$\neg\exists_{n' \notin CH(n),sta}(\mathrm{st}, n', n, sta) \in m \tag{6.7}$$

  By (6.6) and (6.7), n receives no state updates:

  $$\neg\exists_{n',sta}(\mathrm{st}, n', n, sta) \in m \tag{6.8}$$

  It was assumed no node receives a command:

  $$\neg\exists_{n',n'',com}(\mathrm{co}, n', n'', com) \in m \tag{6.9}$$

By (6.6), (6.3) and (6.9), n does not change state during m:

$$\neg\exists_{x,x',sta\neq sta'}((\mathrm{st},n,x,sta)\in m \wedge (\mathrm{st},n,x',sta')\in m) \qquad (6.10)$$

By (6.1), a node only sends a state update if it changed state, it received a command or it received a state update.

By Assumption 6.1.2 and (6.3), n is local loop free, so if there is a state change of n in m, then there must also be a command to or from n in m or a state update of a child of n in m.

By (6.1), (6.8), (6.9) and (6.10), n sends no state updates:

$$\neg\exists_{n',sta}(\mathrm{st},n,n',sta)\in m$$

Hence, $q(i+1)$ holds.

Hence we can conclude that no node sends a state update: for all i, $q(i)$ holds, meaning that there are no n', n'' and sta such that $(\mathrm{st},n',n'',sta)\in m$. If there are neither commands nor state update messages in m, m is empty, which contradicts the assumption that there is a non-local loop ℓ with a non-empty m. Hence, every non-local loop contains at least one command:

$$\exists_{n',n'',com}(\mathrm{co},n',n'',com)\in m$$

∎

Note that in the current CFSM implementation, after a node receives a command, it always sends back a state update. Hence, if there is a command in m, then there is a state update in m as well:

$$(\exists_{n,n',com}(\mathrm{co},n',n,com)\in m))\Rightarrow\exists_{sta}(\mathrm{st},n,n',sta)\in m$$

6.2 General non-local loops

To detect non-local loops, bounded model checking can be used. *Bounded model checking* (BMC) [BCC$^+$03] is a technique in which the possible behaviour of a system is unfolded and then checked for a violation of the property to be checked within a number, say r, of steps. If no violation is found, r is increased up to a certain upper bound u. If one proves that if a violation is possible it must occur within u steps, and the system cannot violate the property in u steps, the violation is not possible and the property holds. BMC is often implemented using SAT tools.

Unfortunately, a general upper bound u to the number of steps is the size of the complete CFSM state space of a system. To prove this claim, we give an example of a state-changing non-local loop that traverses the entire state space before reaching the initial configuration again.

Example Consider a system s as shown in Figure 6.4: consisting of n nodes in a chain structure. The source node 1 is node of CFSM class A and has a chain of children of class B. Node n is a leaf of class C. The CFSM classes are shown in Listing 6, 7 and 8, where i is an integer variable and m is an integer constant.

Intuitively, these systems behave as endless analog counters, where each of the $n-1$ digits traverse its m states, before getting reset by its parent.

We construct a non-local loop for s by starting in the configuration in which every node is in state L1 except node n, that is in state LM[1]:

```
(1: L1, 2: L1, ..., n-2: L1, n-1: L1, n: Lm)
```

In the entire system, only the WHEN clause in node $n-1$ is enabled, so node $n-1$ moves to state L2:

```
(1: L1, 2: L1, ..., n-2: L1, n-1: L2, n: Lm)
```

We skip a few steps in which $n-1$ traverses all its states, and ends in state LM.

```
(1: L1, 2: L1, ..., n-2: L1, n-1: Lm, n: Lm)
```

The only WHEN clause in s that is enabled is in node $n-2$, so node $n-2$ sends a RESTART command to node $n-1$ and waits:

```
(1: L1, 2: L1, ..., n-2: L1, n-1: (Lm, [RESTART]), n: Lm)
```

Node $n-1$ processes the command by sending a RESTART command to node n and moves to state L1. This releases node $n-2$ from waiting, so it completes its action by moving to state L2:

```
(1: L1, 2: L1, ..., n-2: L2, n-1: L1, n: Lm)
```

We skip the steps in which nodes 2 to $n-1$ traverse all their states, progressing when their child is in state LM. Finally s reaches the configuration where all nodes are in state LM:

```
(1: Lm, 2: Lm, ..., n-2: Lm, n-1: Lm, n: Lm)
```

In this configuration, only the WHEN clause in node 1 is enabled. Every node sends a RESTART command to its child, waits for confirmation and moves to state L1 again. This brings s back to the intitial configuration:

```
(1: L1, 2: L1, ..., n-2: L1, n-1: L1, n: Lm)
```

Because in this trace node 1 receives no commands, node n does not change state without receipt of a command, and the initial configuration is not encountered twice, the trace denotes a non-local loop. Every possible configuration appears in this trace, so the non-local loop it denotes traverses the entire state space.

This proves that there are loops that traverse the complete state space and hence there are systems that require the whole state space to be traversed while checking for non-local loops using bounded model checking. Even our smallest system, ALICE, has a state space of 10^{12052}. A usable implementation of a BMC check for general non-local loops on these state spaces is infeasible. To decrease the size of the state spaces, we apply reductions.

[1]For readability reasons, we ommit command queues when they are empty

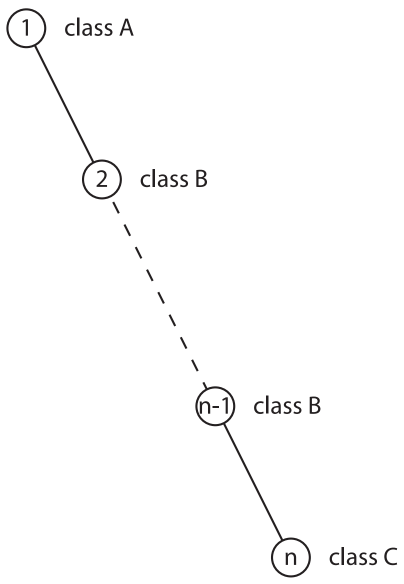


Figure 6.4: A system s, consisting of a chain of nodes $1 \cdots n$

6.3 Reductions

In this section, we describe how we reduced state space sizes. We first describe a reduction that uses absence of certain FSM constructs to remove nodes: top bouncer reduction. Several separate systems can result from applying this reduction. Therefore, we describe another reduction to remove duplicates from separate systems: duplicate system reduction.

6.3.1 Top bouncer reduction

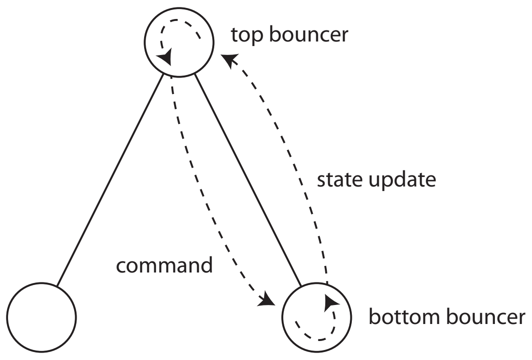


Figure 6.5: Schematic overview of two necessaries for non-local loops: a top bouncer 'bounces' a state update back as a command and a bottom bouncer bounces a command back as a state update.

```
class: A
    state: Li !(for 1 <= i < m)
        when ($ALL$FwCHILDREN in_state Lm) do TICK
        action: TICK
            do RESTART $ALL$FwCHILDREN
            wait ( $ALL$FwCHILDREN )
            move_to L(i+1)

    state: Lm
        when ($ALL$FwCHILDREN in_state Lm) do RESTART
        action: RESTART
            do RESTART $ALL$FwCHILDREN
            wait ( $ALL$FwCHILDREN )
            move_to L1
```

Listing 6: Class A

```
class: B
    state: Li !(for 1 <= i < m)
        when ($ALL$FwCHILDREN in_state Lm) do TICK
        action: TICK
            do RESTART $ALL$FwCHILDREN
            wait ( $ALL$FwCHILDREN )
            move_to L(i+1)

    state: Lm
        action: RESTART
            do RESTART $ALL$FwCHILDREN
            wait ( $ALL$FwCHILDREN )
            move_to L1
```

Listing 7: Class B

Top bouncers In a CFSM system, commands go down the graph from the sources and state updates go up the graph towards the sources. If all nodes in the graph are free of local loops, the only way an infinite sequence of commands and state updates, a non-local loop, can happen, is when a command somehow triggers a state update and a state update somehow triggers a command. This can happen in top and bottom bouncers. Figure 6.5 shows an intuition.

Definition 6.3.1 *A* candidate top bouncer *is an FSM construct that upon receipt of a state update might send a command: an action clause containing a* DO *statement mentioned in a* DO *referrer.*

The example below shows such a construct:

Example

```
state: ON
    when ( $ANY$FwCHILDREN in_state OFF ) do GO_OFF
    action: GO_OFF
        do OFF $ALL$FwCHILDREN
```

```
class: C
    state: Lm
```

Listing 8: Class C

Note that a language construct of the described shape is not guaranteed to actually send a command, as the DO statement might be inside an IF statement or the guard of the WHEN clause might be false in some configurations.

Definition 6.3.2 *A candidate top bouncer is a* top bouncer *when it, in a production or simulated system, actually sends a command.*

Say the example above is present in a node n. Then the candidate top bouncer becomes a top bouncer if, in a running system, n is in state ON and has a child that is in state OFF. Notationwise, $cTB(n)$ holds if and only if node n has a candidate top bouncer, and $TB(n, \ell)$ holds if and only if a top bouncer in node n sends a command during non-local loop ℓ. Obviously, for all nodes n and non-local loops ℓ it holds that $TB(n, \ell)$ implies $cTB(n)$.
Given a non-local loop ℓ, no way was found to determine whether $TB(n, \ell)$ holds for a certain node n without requiring knowledge of the system involved and a behaviour reconstruction. However, in the analyses following, $TB(n, \ell)$ is shown to be required for non-local loops. We first prove that a node only sends a command if it received a command or a top bouncer is active.

Lemma 6.3.3 *For every node n, n cannot send a command to one if its children unless either:*

- *n received a command from one of its parents*
- *n received a state update from one of its children and after that a top bouncer in n sent a command*

Proof by inspection of FSM constructs:
The only FSM instruction that can send a command is a DO statement. The only place a DO statement can appear is in an action clause, which is executed upon receipt of a command or by a DO referrer. We apply case distinction over these two options:

- n can send a command if it received a command from one of its parents.
- A DO referrer is only called if the when clauses are evaluated. Evaluation of a when clause only happens when n receives a state update from one of its children. By definition, a top bouncer sends a command upon receipt of a state update.

Hence, node n only sends a command to one of its children if n received a command from one of its parents or it received a state update from one of its children and after that a top bouncer in n sent a command. ∎

From this lemma, it follows that for every non-local loop ℓ and for all nodes n, n only sends a command if it receives a command, or receives a state update and a top bouncer is active:

$$\begin{aligned}\forall_{\ell=(ac_1,m),n}((\exists_{n'\in CH(n),com}(\text{co},n,n',com)\in m)\Rightarrow\\((\exists_{n'\in P(n),com}(\text{co},n',n,com)\in m)\vee\\(\exists_{n'\in CH(n),sta}(\text{st},n',n,sta)\in m\wedge TB(n,(ac_1,m)))))\end{aligned}$$

where ℓ is a non-local loop.

Using the lemma, we can prove that in every non-local loop a top bouncer is active.

Theorem 6.3.4 *For every non-local loop ℓ, there is a node n such that $TB(n,\ell)$ holds: $\forall_\ell\exists_n TB(n,\ell)$, where ℓ is a non-local loop.*

Proof Towards contradiction, suppose there is a system s having a non-local loop $\ell=(ac_1,m)$, such that there is no top bouncer active in ℓ:

$$\neg\exists_{n'}TB(n',\ell) \tag{6.11}$$

Let k be a list containing all nodes of s. Assume k is sorted in a topological order: if node n is a parent of node n', then n must appear in k before n'. As s is a directed acyclic graph, such an ordering exists. Define $r(i)$ to be true if and only if the node n at position i in k sends no commands during ℓ:

$$r(i)\equiv\neg\exists_{n',com}(\text{co},n,n',com)\in m$$

We prove by induction that $r(i)$ holds for all nodes in k.

Base case: $r(0)$.
Let n be the node in k at position 0. Observe that n cannot have parents and therefore is a source: $n\in S(s)$.

As $n\in S(s)$, by definition of ℓ, n will not receive a command:

$$\neg\exists_{n',com}(\text{co},n',n,com)\in m \tag{6.12}$$

By Lemma 6.3.3, (6.12) and (6.11), n sends no commands:

$$\neg\exists_{n',com}(\text{co},n,n',com)\in m$$

Hence, $r(0)$ holds.

Induction step: assume as the induction hypothesis that $r(j)$ holds for all j smaller than or equal to i. Let n be the node referenced in k at position $i+1$. Perform a case distinction on whether n is a source:

– Case: n is a source: $n\in S(s)$
Analogous to the proof of the base case: $r(i+1)$.

– Case: n has at least one parent: $n\notin S(s)$
By the ordering of k, all parents of n are referenced in k at positions j with $j\le i$. By the induction hypothesis, no parent of n sends a command:

$$\forall_{n'\in P(n)}\neg\exists_{n'',com}(\text{co},n',n'',com)\in m) \tag{6.13}$$

n can only receive commands from its parents:

$$\neg\exists_{n'\notin P(n),com}(\text{co}, n', n, com) \in m \tag{6.14}$$

By (6.13) and (6.14), n receives no commands:

$$\neg\exists_{n',com}(\text{co}, n', n, com) \in m \tag{6.15}$$

By Lemma 6.3.3, (6.15) and (6.11), n sends no commands:

$$\neg\exists_{n',com}(\text{co}, n, n', com) \in m$$

Hence, $r(i+1)$ holds.

It hence follows that $r(i)$ holds for all i, which means that there exist no n, n' and com such that $(\text{co}, n, n', com) \in m$. This contradicts Lemma 6.1.3, so ℓ cannot be a non-local loop or there must be a top bouncer is s. This proves that for every non-local loop ℓ, there is a node n such that $TB(n, \ell)$ holds. ∎

Reduction By definition a top bouncer is a language construct that upon receipt of a state update sends a command. By Theorem 6.3.4, in every non-local loop at least one top bouncer is active. We prove a stronger version, stating that a source without top bouncer can be removed without altering non-local loop presence in the other parts of the system, and use that to reduce the system.

Theorem 6.3.5 *Assume two systems s and s', such that s' is obtained from s by removing a source n. Suppose n cannot have top bouncers: there exists no non-local loop ℓ'' such that $TB(n, \ell'')$. Then there exists a non-local loop $\ell = (ac_1, m)$ in s if and only if there exists a non-local loop $\ell' = (ac'_1, m')$ in s'.*

Proof We prove both directions of the bi-implication separately:

- Assume system s' has a non-local loop $\ell' = (ac'_1, m')$. Then a non-local loop (ac_1, m) for s can be constructed by replaying m' in system s. If node n is both a source and a leaf, and is therefore not connected to the other nodes, this construction is trivial. Therefore, in the following assume that n is not a leaf.

  Before we give the construction, we prove that node n sends no commands during any non-local loop $\ell'' = (ac''_1, m'')$ for system s. As $n \in S(s)$, n receives no commands:

  $$\neg\exists_{n',com}(\text{co}, n', n, com) \in m'' \tag{6.16}$$

  By (6.16), $\neg TB(n, \ell'')$ and Lemma 6.3.3, n sends no command during m'':

  $$\neg\exists_{n',com}(\text{co}, n, n', com) \in m'' \tag{6.17}$$

  Hence, n sends no commands during any non-local loop ℓ''.

  By construction, each child n' of n is present in both s and s'. By (6.17), n' receives no commands from n when in s. Moreover, if $n' \in S(s')$, by (6.16) n' receives no commands either. Therefore no child of n receives a command from n in s nor s'. Hence, no node in s' can distinguish between being in s and s' and hence every node can perform the same behaviour in s' as in s. With this, we construct the non-local loop ℓ:

1. Start with s in annotated configuration ac'_1. This leaves node n, which is put in an arbitrary state. As n receives no commands, choose the command queue empty. Now system s is in a valid annotated configuration. Call this annotated configuration acx_0:

   $$acx_0 = ac'_1[n \to (sta_0, [])]$$

   where sta_0 is an arbitrary state of n.

2. Set $i = 0$.

3. Replay m' in s, recording all sent messages in m_i. Call the resulting annotated configuration acx_{i+1}:

   $$acx_i \xrightarrow{m_i} acx_{i+1}$$

   As all nodes, except n, can perform the same behaviour in s and s', this replay is possible. As $ac'_1 \xrightarrow{m} ac'_1$ and n receives no commands, acx_{i+1} is equal to $ac'_1[n \to (sta_{i+1}, [])]$ for some state sta_{i+1}.

4. Repeat this last step, increasing i, until a resulting annotated configuration is encountered a second time. Formally stated: until an a and b exist, such that $a < b$ and $acx_a = acx_b$. This will happen in at most $|n.states|$ steps.
   Then a non-local loop for s is $(acx_a, [m_a +\!\!+ \cdots +\!\!+ m_{b-1}])$, as s can start in annotated configuration acx_a, run and produce the messages in m_a to m_{b-1} without passing through acx_a, and end up in acx_a again.

By construction, no leaf in s' changes state infinitely often in ℓ without receipt of a command infinitely often. We assumed that n is not a leaf, so no leaf in s changes state infinitely often without receipt of a command. Hence, ℓ is a non-local loop.

Given an arbitrary non-local loop ℓ' for s', a non-local loop ℓ for s can be constructed. Hence, if s' contains a non-local loop, then s contains a non-local loop.

- To prove the other direction of the bi-implication, assume towards contradiction that s' has no non-local loop and s has a non-local loop $\ell = (ac_1, m)$.

  $n \in S(s)$, so by definition of ℓ, n receives no command in m:

  $$\neg\exists_{n',com}(\mathrm{co}, n', n, com) \in m \tag{6.18}$$

  By (6.18), $\neg TB(n, \ell)$ and Lemma 6.3.3, n sends no command in m:

  $$\neg\exists_{n',com}(\mathrm{co}, n, n', com) \in m \tag{6.19}$$

  As the children of n receive no commands from n, these children cannot distinguish between being in s and s'. Therefore, no node in s' can distinguish between being in s and s'. As there is no repeatable m' for s', there is no repeatable m for these nodes in s either. Hence, there is no command sent to or from any of these nodes in m:

$$\forall_{n' \in s'} \neg \exists_{n'',com} ((\text{co}, n', n''com) \in m \vee (\text{co}, n'', n'com) \in m) \tag{6.20}$$

By (6.18), (6.19) and (6.20), m does not contain any commands, which contradicts Lemma 6.1.3.

Hence, if s contains a non-local loop, then s' contains a non-local loop.

Hence, s contains a non-local loop if and only if s' contains a non-local loop. ∎

To make the reduction more powerful, we prove that a system consisting of a single node cannot have a non-local loop.

Theorem 6.3.6 *A system s consisting of a single node n cannot have a non-local loop.*

Proof Towards contradiction, suppose s has a non-local loop $\ell = (ac_1, m)$. As n has neither parents nor children, n is both a source and a leaf: $n \in S(s) \wedge n \in L(s)$. By definition of ℓ, n receives no commands:

$$\neg \exists_{n',com} (\text{co}, n', n, com) \in m \tag{6.21}$$

As $CH(n) = \emptyset$, by (6.21) and Lemma 6.3.3, n sends no commands:

$$\neg \exists_{n',com} (\text{co}, n, n', com) \in m$$

If n receives and sends no commands, m contains no commands, which contradicts Lemma 6.1.3. Hence, a system consisting of a single node cannot contain a non-local loop. ∎

As for all n it holds that $\neg cTB(n) \Rightarrow \neg \exists_\ell TB(n, \ell)$, Theorem 6.3.5 can be applied by searching for sources without candidate top bouncers and removing them. This procedure can be repeated as long as there are sources without candidate top bouncers left. If a source is encountered that has no children, Theorem 6.3.6 allows it to be removed.

If a source is removed, its subsystems might become independent. It is advantageous to check for independence and store the independent subsystems separately for further analyses, as the state spaces of the subsystems then are summed instead of multiplied, yielding a significantly smaller state space. This independency check is performed by converting the system structure to an undirected graph, where two nodes are connected if and only if they have a parent-child relation. The resulting graph is then checked for connected components using a standard library. Then each connected component is an independent system, since no node in it has a parent-child relation with any node in any other connected component.

Tool *Top bouncer reduction* is the procedure of applying the two mentioned reductions and splitting the resulting independent subsystems. A tool was written that performs top bouncer reduction, its pseudocode is given in Listing 9. Given a list of sources, the algorithm performs the reductions. It keeps a set *queue* of sources that need to be processed, and a set *result* of unreducable sources. Invariant is that $queue \bigcup result$ remains a list of all sources of the ((partially) reduced) system. In each step, a source is either removed or moved to *result*. Figure 6.6 summarises what happens when a node is encountered that can be reduced: node 2 is removed, node 3 remains a child of node 4 and node 2 becomes a source . Node 2 could be reducable and is added to *queue*. Figure 6.7 shows what happens when a node is encountered that cannot be reduced: it is simply moved from *queue* to *result*. Theorem 6.3.6 allows us to remove any source that has no children.

The algorithm ends when $queue = \emptyset$. An empty *queue* implies that there are no more nodes that can be removed. Finalisation is easily proven by using as a variant function the number of nodes reachable from the sources in *queue* by only following parent-child relations downwards. Top bouncer reduction finishes by separating the independent subsystems. The result of this last function is put in *resultSystems*.

```
function topBouncerReduction(sources)
    $queue, result \leftarrow sources, []$
    while pick a $node \in queue$ and remove it from $queue$ do
        if $node$ has children then
            if $node$ has a candidate top bouncer then                ▷ Figure 6.7
                $result \leftarrow result \bigcup \{node\}$
            else                                                      ▷ Figure 6.6
                for $child \in node.children$ do
                    Remove $node$ as a parent of $child$
                    if $child$ has no parents left then
                        $queue \leftarrow queue \bigcup \{child\}$
                    end if
                end for
            end if
        end if
    end while
    $resultSystems \leftarrow getIndependentSystems(result)$
end function
```

Listing 9: Top bouncer reduction, where $getIndependentSystems()$ is a function that given a list of sources, provides the longest list of systems, such that every system is independent of all the others.

Results Table 6.1 shows the results of top bouncer reduction. As 'before', we took the systems as reduced by reachability reduction. The number of nodes here includes leaves, as leaves contribute to state space. We conclude from the data that top bouncer reduction can make a huge difference: in CMS and in particular ATLAS, state spaces are drastically reduced. Investigation of the ATLAS system revealed that only one CFSM class in the ATLAS DCS has a top bouncer. ATLAS told us they avoid using the constructs that we defined as top bouncers.

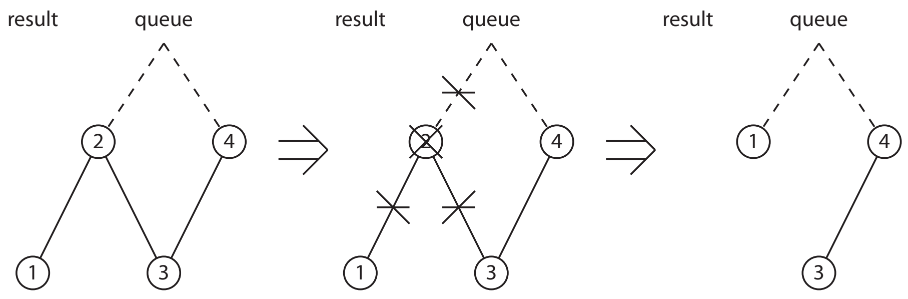


Figure 6.6: Top bouncer reduction: node 2 gets removed. Node 1 has no parents left and is added to *queue*. Left: situation before, middle: node 2 is removed from all its parents and children, right: situation after.

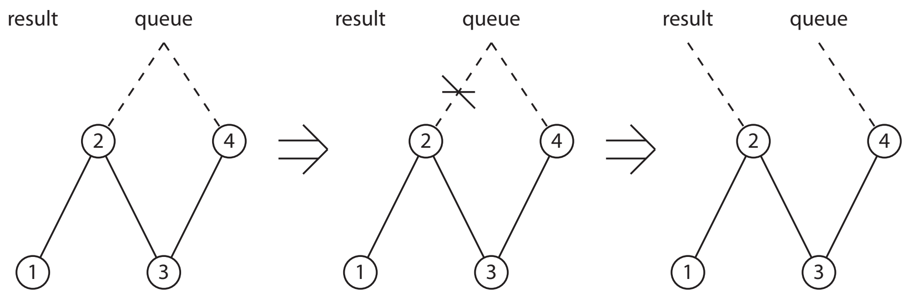


Figure 6.7: Top bouncer reduction: node 2 cannot be removed. It is moved from *queue* to *result*. Left: situation before, middle: node 2 is removed from *queue*, right: situation after.

		Nodes		State space		systems
		before	-after	before	-after	after
CMS	11-2011	32356	- 15243	10^{26400}	- 10^{1191}	488
	01-2012	32388	- 15245	10^{26186}	- 10^{1194}	488
	03-2012	32480	- 15245	10^{26276}	- 10^{1194}	488
	05-2012	32724	- 15245	10^{26368}	- 10^{1192}	488
LHCb	11-2011	79519	- 78203	10^{58325}	- 10^{57477}	3
ATLAS	05-2012	76683	- 736	10^{58922}	- 10^{36}	16
ALICE	05-2012	14996	- 9980	10^{12052}	- 10^{3351}	212

Table 6.1: Results of top bouncer reduction

6.3.2 Children-specific top bouncer reduction

In a CFSM system structured as a tree, commands go down the graph from the sources and state updates go up the graph towards the sources. After applying TBR, every source contains a candidate top bouncer. However, if the definition of top bouncers is extended with the information to *which* children the top bouncer sends a command, the reduction can be extended as well: if a child c receives no commands and its subsystem is non-local loop free, it will eventually stabilise. This is the same behaviour as if c and all its children would be replaced by a leavified version of c. A *leavified* version of node c is a leaf containing the states of c, without their WHEN and ACTION clauses. To summarise: if a subsystem receives no commands, it can be replaced by a leavified version of its source and checked for non-local loops in isolation.

This idea of *children-specific top bouncer reduction* was implemented for tree-structured systems. As we discovered the systems were structured like directed acyclic graphs, determining whether children-specific top bouncer reduction is applicable to directed acyclic graphs needs to be subject of further study.

6.3.3 Duplicate system reduction

Top bouncer reduction can produce independent systems. Node names and identifiers have no influence on behaviour of nodes (no guard can select on node names or identifiers), so if two systems are equal up to renaming of nodes, only one of them needs to be checked for non-local loops.

Consider two independent systems s and s'. Then they can be determined to be equal as follows: Construct a directed acyclic graph G, having as nodes the nodes of s, coloured with the CFSM class they are instances of. A directed edge (n, n') is added to G if and only if $n' \in CH(n)$. Do the same for s' constructing G'. Then s and s' can perform the same behaviour if G and G' are isomorphic respecting colours. The isomorphism test used in the tool is provided by the NetworkX package for Python, version 1.6. As the isomorphism test is expensive, in particular when the two graphs are not isomorphic, the tool generates a simple hash of the two systems. If the two hashes are different, the isomorphism test is not applied. In the systems tested, this filters out all isomorphic tests that would have returned non-isomorphism. Listing 10 shows the straightforward pseudocode of duplicate system reduction.

Results The results of this *duplicate system reduction* tool on the systems of CMS, LHCb and ATLAS are in Table 6.2. ‘Before’ refers to the systems remaining after top bouncer reduction. The data show that state space does not decrease much by applying duplicate system reduction, but the number of systems to be checked decreases by roughly a factor 10 in most experiments.

6.4 State-keeping non-local loops

Even though bottom bouncer reduction and duplicate system reduction reduce the state spaces, even a non-local loop check on the system of ATLAS, with its reduced state space of 10^{35}, is infeasible. Therefore, we identified a special case of a non-local loop that can be detected using bounded model checking with an

```
function DUPLICATESYSTEMREDUCTION($systems$)
    $graphs = []$
    for $system \in systems$ do                              ▷ Create a graph for each system
        $graphs[system] \leftarrow$ create a new directed graph
        for $node \in system$ do
            add $node$ as a node to $graphs[system]$
        end for
        for $node \in system$ do
            for $child \in node.children$ do
                add edge ($node$, $child$) to $graphs[system]$
            end for
        end for
    end for
    $result \leftarrow \{\}$                                  ▷ Get graphs without duplicates
    for $system \in systems$ do
        $found \leftarrow false$
        for $system2 \in result$ do
            if $hash(system1) = hash(system2)$ then
                if $isomorphic(graphs[system], graphs[system2])$ then
                    $found \leftarrow true$
                    break
                end if
            end if
        end for
        if $\neg found$ then
            $result \leftarrow result \bigcup \{system\}$
        end if
    end for
end function
```

Listing 10: Duplicate system reduction, where *isomorphic*() is a function that given two graphs returns whether they are isomorphic respecting colouring, where colours represent CFSM classes. $hash(s)$, given a system s, returns a hash based on the structure of s.

upper bound of one round: the *state-keeping non-local loop*. This is a non-local loop during which no node changes state.

Definition 6.4.1 *A non-local loop $\ell = (ac_1, m)$ is a state-keeping non-local loop if and only if*

$$\neg\exists_{n,n',sta \neq sta'}((st, n, n', sta) \in m \wedge (st, n, n', sta') \in m) \tag{6.22}$$

As no node is allowed to change state during the loop, obviously no leaf is allowed to change state during the loop either. This happens often in practice: hardware might need some time to change its state after receipt of a command. Meanwhile, the CFSM control system could loop and flood the computing network with messages, filling up queues and possibly causing genuine command messages to be dropped. Therefore, in this section it is assumed leaves do not change state during the loop:

		Nodes		Systems		State space	
		before	-after	before	-after	before	-after
CMS	11-2011	15243	- 5591	488	- 49	10^{1191}	- 10^{1191}
	01-2012	15245	- 5593	488	- 49	10^{1194}	- 10^{1194}
	03-2012	15245	- 5593	488	- 49	10^{1194}	- 10^{1194}
	05-2012	15245	- 5593	488	- 49	10^{1192}	- 10^{1192}
LHCb	11-2011	78203	- 78203	3	- 3	10^{57477}	- 10^{57477}
ATLAS	05-2012	736	- 46	16	- 1	10^{36}	- 10^{35}
ALICE	05-2012	9980	- 7522	212	- 16	10^{3351}	- 10^{3351}

Table 6.2: Results of duplicate system reduction

Assumption 6.4.2

$$\neg\exists_{n\in L(s),n',sta\neq sta'}((st,n,n',sta)\in m \wedge (st,n,n',sta')\in m)$$

where s is a system and $\ell = (ac_1, m)$ is a non-local loop in s.

6.4.1 Translation

To detect state-keeping non-local loops, we introduce a translation from a system to a satisfiability problem. In this section we give this translation and prove that it is correct.

By Theorem 6.3.4, in every non-local loop at least one top bouncer is involved, so assume a system s having a non-empty set of nodes having candidate top bouncers *topBouncers*. In this section, several properties are used as objects and given in dot-notation. For instance *state.whenClauses* points to the list of WHEN clauses belonging to *state*. The set of names of all states of all nodes in s is *allStates* and *allCommands* is the set of all command names.

Variables

- $nodeInState_{n,sta}$ denotes whether node n is in state sta.
- $commandSent_{com,n,n'}$ denotes whether command com is sent from node n to node n'.
  By definition of non-local loops, sources receive no commands. Therefore, in contrast to the (co, n, n', com) tuples used before, we can safely require both n and n' to be nodes, and not grandparents.
  Note that this abstracts from how many times a command is sent. This is fine as no node changes its state, so a command sent twice will have the same effect twice. This also renders the message queue size abstraction in Section 6 irrelevant.

Helper functions

- $checkWhenClauses(node, whenClauses)$ produces a formula that is equivalent to true if and only if both:

- If a WHEN clause is enabled, execution of neither its referrer nor the statements it points to change the state of *node* ($stateIsKept(node, whenClause.referrer)$ is true)
- If a WHEN clause is enabled, for every command that is sent by it, the corresponding *commandSent* variable must be true. ($commandsAreSent(node, whenClause.referrer)$ is true)

- $stateIsKept(node, constructs)$ produces a formula that is equivalent to true if and only if *node* does not change its state when the *constructs* are executed. If *constructs* is a DO referrer, the statements in the ACTION clause it points to are considered.

- $commandsAreSent(node, constructs)$ produces a formula that is equivalent to true if and only if for every command that is sent by *constructs*, the corresponding *commandSent* variable is true. If *constructs* is a DO referrer, the statements in the ACTION clause it points to are considered.

- $oneCommandSent(node, statements)$ produces a formula that is equivalent to true if and only if at least one command is sent when the *statements* are executed according to the CFSM specification.

Formula First, the possible behaviour of the system is modelled.
For every node n:

- n is in one defined state:

$$\bigvee_{sta \in n.states} \left(\left(\bigwedge_{sta' \in allStates \setminus \{sta\}} \neg nodeInState_{n,sta'} \right) \wedge nodeInState_{n,sta} \right) \tag{6.23}$$

- The WHEN clauses of n do not change the state of n and commands are sent correctly:

$$\bigwedge_{sta \in n.states} \big(nodeInState_{n,sta} \Rightarrow checkWhenClauses(n, sta.whenClauses) \big) \tag{6.24}$$

- When n is not a source,[2] n does not change state as a result of a command received. Moreover, on receipt of a command, commands are sent

[2]This is ensured by the translation tool.

correctly:

$$\bigwedge_{sta \in n.states} \Bigg(nodeInState_{n,sta} \Rightarrow \Bigg(\bigwedge_{a \in sta.actions} \Bigg(\bigvee_{n' \in n.parents} commandSent_{a,n',n} \Bigg) \Rightarrow \big(stateIsKept(n, a.statements) \wedge commandsAreSent(n, a.statements) \big) \Bigg) \Bigg) \tag{6.25}$$

Second, at least one top bouncer tb must be enabled:

- The node that contains the candidate top bouncer, is in the state of the candidate top bouncer:

$$nodeInState_{tb.node,tb.state} \tag{6.26}$$

- The guard of the candidate top bouncer is true and all preceding guards are false:

$$tb.guard \wedge \bigwedge_{g \in tb.state.guards \wedge g \; precedes \; tb.guard} \neg g \tag{6.27}$$

- At least one command is sent by the action clause of the candidate top bouncer:

$$oneCommandSent(tb.node, tb.action.statements) \tag{6.28}$$

The complete formula f is the combination of the formulae mentioned here:

$$f = \bigwedge_{n \in s.nodes} ((6.23) \wedge (6.24) \wedge (6.25)) \wedge \bigvee_{tb \in topBouncers} ((6.26) \wedge (6.27) \wedge (6.28))$$

f is translated to SMT by a straightforward script. This SMT can then be solved by a SMT solver, such as Yices [DMdM06]. In case f is satisfiable, evidence is generated consisting of an assignment to all variables. Using this evidence, we show that s has a state-keeping non-local loop if and only if f is satisfiable.

Theorem 6.4.3 *f is satisfiable if and only if there is a state-keeping non-local loop in the system s.*

Proof We prove both directions of the bi-implication separately:

- Suppose f is satisfiable.
  Clause (6.23) of f guarantees that for every node n in s, there is exactly one sta of the states of n for which $nodeInState_{n,sta}$ holds. Combining these gives a configuration c_1 for s. The variables $commandSent_{com,n,n'}$ denote whether a *com* command is sent. The number of command messages sent was abstracted from and the order in which the commands are sent is not

known, so constructing a message list m requires a simulation of s, but it is obvious this is possible. The same holds for transforming the valid configuration c_1 into an annotated configuration ac_1.

Having a valid and complete annotated configuration ac_1 and a valid message trace m, in order to prove (ac_1, m) to be a state-keeping non-local loop, there is left to prove:

- No node changes state.
  By Clauses (6.24) and (6.25), no node changes state during the loop:

$$\neg\exists_{n,n',sta\neq sta'}((\mathrm{st}, n, n', sta) \in m \wedge (\mathrm{st}, n, n', sta') \in m) \qquad (6.29)$$

- The sources of s receive no commands.
  When we defined the $commandSent_{com,n,n'}$ variable, we excluded grandparents from sending commands. Sources receiving commands from other nodes is excluded by a correct implementation of the $commandsAreSent(node, statements)$ helper function. Hence, the sources of s receive no commands.
- The leaves of s do not change state during the loop without receiving a command in the loop.
  This is implied by (6.29).

Hence, if f is satisfiable, there is a state-keeping non-local loop in s.

- Suppose there exists a state-keeping non-local loop $\ell = (ac_1, m)$ for system s.
  By (6.22), no node in s changes state during ℓ. Hence, system s is in only one configuration c_1 during ℓ. This c_1 can be straightforwardly translated to an assignment of the variables $nodeInState_{n,sta}$. The same holds for translating trace m into an assignment to the variables $commandSent_{com,n,n'}$. Call the resulting assignment a. Left to prove: a satisfies f.

  As ac_1 is a valid annotated configuration for s, Clause (6.23) holds for all nodes n. For every node n, Clause 6.27 consists of two parts: the state of n does not change as a result of executing a WHEN clause (which is implied by (6.22)), and the commands that should be sent by executing this WHEN clause are correctly denoted in the $commandSent$ variables (which is implied by a correct implementation of the helper function $commandsAreSent$). For every node n, Clause (6.25) consists of two parts: n does not change as a result of a command (which is implied by (6.22)), and the commands that should be sent on receipt of a command are correctly denoted in the $commandSent$ variables (which is implied by a correct implementation of the helper function $commandsAreSent$). Hence, the first part of f holds for a:

$$\bigwedge_{n\in s.nodes} ((6.23) \wedge (6.24) \wedge (6.25))$$

  As ℓ is a non-local loop, by Theorem 6.3.4 there exists a node n such that $TB(n, \ell)$ holds, meaning that there is a top bouncer in ℓ. It is not hard to reason that this implies that the second part of f holds for a:

$$\bigvee_{tb \in topBouncers} ((6.26) \wedge (6.27) \wedge (6.28))$$

Hence, if there is a state-keeping non-local loop in s, then f is satisfiable.

Hence, f is satisfiable if and only if there is a state-keeping non-local loop in s. ∎

Before we give a description of the tool we used to verify the DCS software systems of the LHC, we introduce a reduction.

6.4.2 Bottom bouncer reduction for state-keeping non-local loops

We introduced top bouncer reduction in Section 6.3.1 and briefly mentioned bottom bouncers. In this section, we give a definition of bottom bouncers and use a special kind of bottom bouncer to reduce the state space.

Bottom bouncers A *candidate bottom bouncer* is similar to a candidate top bouncer: an FSM construct in a node that upon receipt of a command sends a state update. A *bottom bouncer* is a candidate bottom bouncer in a running system that actually sends a state update. In the CFSM implementation, a node always sends a state update upon receipt of a command, so every node has candidate bottom bouncers. Therefore, we make a distinction: a *state-changing bottom bouncer* is a bottom bouncer that upon receipt of a command not only sends a state update, but also changes state. A *state-keeping bottom bouncer* is the opposite: it does not change state.

Define a *candidate state-changing bottom bouncer* to be an FSM construct that upon receipt of a command might send a state update and change state. It is not hard to find the only shape a candidate state-changing bottom bouncer can have: a MOVE_TO statement in an action clause.

Having the definition of state-changing bottom bouncers, we introduce the reduction.

Reduction Consider a parent-children combination of a node n with, as its children, some leaves. If the behaviour of n is indistinguishable from the behaviour of a leaf having the same states as n, we might as well replace n with a leaf having the same states as n; a leavified version of n.

Theorem 6.4.4 *Assume two systems s and s', such that s' is obtained from s by replacing a parent-children node combination $n(l_1 \dots l_k)$ with a single leavified version of n: n'. Suppose $n(l_1 \dots l_k)$ contains no state-keeping non-local loop, suppose that n has no candidate state-changing bottom bouncer and suppose that nodes $l_1 \dots l_k$ are leaves having no other parents than n. Then s contains a state-keeping non-local loop if and only if s' contains a state-keeping non-local loop.*

Proof As n contains no state-changing bottom bouncers, n will not change state on receipt of a command. By Assumption 6.4.2 n' and $l_1 \dots l_k$ do not change state at all.
We prove the directions of the bi-implication separately:

- Assume s has a state-keeping non-local loop. As $n(l_1 \dots l_k)$ contains no state-keeping non-local loop, there must be a top bouncer active in one of the other nodes of s. We are in a state-keeping non-local loop, so n does not change state. Therefore, the other nodes of s cannot distinguish between being in s or in s'. Hence, s' contains a state-keeping non-local loop.

- Assume s' has a state-keeping non-local loop. We first prove that n cannot change state during the loop often by examining the FSM MOVE_TO constructs:

  - MOVE_TO statement
    As n contains no candidate state-changing bottom bouncers, there is no MOVE_TO statement that can be executed.
  - MOVE_TO referrer
    All children of n are leaves, so by Assumption 6.4.2 they will not change state. By Assumption 6.1.2, n contains no local loops, implying that there is no endless sequence of state changes caused by MOVE_TO referrers.

  Hence, n does not change state during the loop, so the other nodes in s cannot distinguish between being in s or in s'. Therefore, s contains a state-keeping non-local loop.

We conclude that s contains a state-keeping non-local loop if and only if s' contains a state-keeping non-local loop. ∎

Tool Theorem 6.4.4 can be applied to all nodes n for which it holds that a) n has no candidate state-changing bottom bouncers, b) n is not a source[3], c) n has children, d) all children of n are leaves with exactly one parent and e) n with its children contains no state-keeping non-local loop. Our tool, described in Listing 11, keeps a list of candidate nodes for which b), c) and d) hold; for an example, see Figure 6.8. For each candidate node n, the tool verifies whether a) holds and if so, performs a reduction on n as shown in Figure 6.9: n is replaced by a leavified version n' of itself. This reduction step might introduce new candidate nodes which are added to the candidate nodes list.

To satisfy e), that states that the combination of n and its children must be state-keeping non-local loop free, n must be checked for state-keeping non-local loops. Our tool makes n and its children a separate system and, as it could be reducible again, adds this system to the queue of top bouncer reduction.

This procedure, *bottom bouncer reduction*, is repeated as long as there are candidate nodes left. As every node can be added to the candidates list once, run time is linear in the number of nodes of s.

[3]The reduction states that we can reduce n if n with its children is state-keeping non-local loop free in isolation. If n is a source, this leads to circular reasoning.

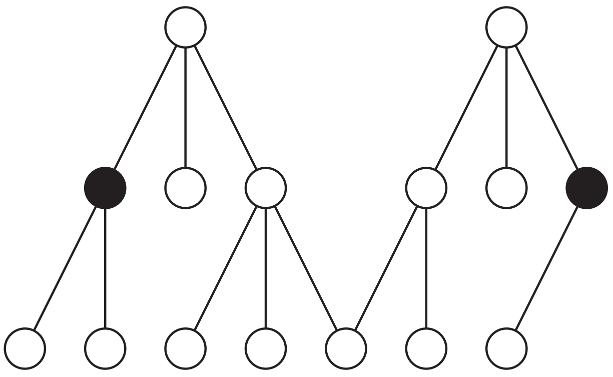


Figure 6.8: Bottom bouncer reduction: a system, of which the candidate nodes are filled black

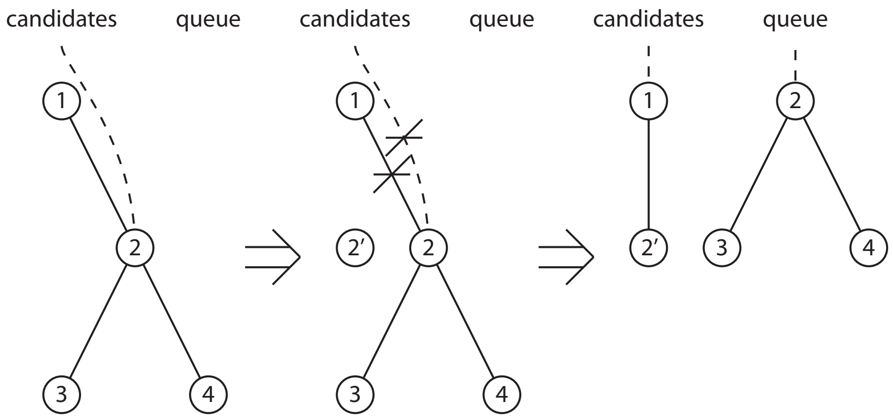


Figure 6.9: Bottom bouncer reduction: node 2 gets leavified. A leavified copy $2'$ is created, node 2 and its children are checked in isolation. After this, node 1 is a candidate. Left: situation before, middle: node 2 is removed from all its parents, right: situation after.

```
function bottomBouncerReduction($sources$: {$node$})
    $candidates \leftarrow$ all nodes for which $isCandidate()$ holds
    $queue \leftarrow \{\}$
    while pick a $node \in candidates$ and remove it from $candidates$ do
        if $node$ has no state-changing bottom bouncers then        ▷ Figure 6.9
            $queue \leftarrow queue \bigcup \{node\}$
            $node' \leftarrow$ leavified version of $node$
            while pick $parent$, a parent of $node$ do
                Remove $parent$ as a parent of $node$
                Add $parent$ as a parent of $node'$
                if $isCandidate(parent)$ then
                    $candidates \leftarrow candidates \bigcup \{parent\}$
                end if
            end while
        end if
    end while
end function
```

Listing 11: Bottom bouncer reduction, where *isCandidate*() is a function that given a node n returns whether all: a) n is not a leaf, b) n is not a source, c) every child of n is a leaf and has only one parent.

		Nodes		Systems		State space	
		before	- after	before	- after	before	- after
CMS	11-2011	5591	- 4872	49	- 45	10^{1191}	- 10^{1189}
	01-2012	5593	- 4874	49	- 45	10^{1194}	- 10^{1191}
	03-2012	5593	- 4874	49	- 45	10^{1194}	- 10^{1191}
	05-2012	5593	- 4874	49	- 45	10^{1192}	- 10^{1189}
LHCb	11-2011	78203	- 78015	3	- 3	10^{57477}	- 10^{57386}
ATLAS	05-2012	46	- 46	1	- 1	10^{35}	- 10^{35}
ALICE	05-2012	7522	- 6912	16	- 15	10^{3351}	- 10^{2848}

Table 6.3: Results of bottom bouncer reduction

Results Table 6.3 shows the results of bottom bouncer reduction. The starting points are the systems as left by the duplicate system reduction tool. The results show that bottom bouncer reduction does not reduce state space drastically, but saves a bit in some systems.

6.4.3 Tool

A tool was developed that performs state-keeping non-local loop detection on a set of reduced systems resulting from bottom bouncer reduction. This tool translates the system to an SMT formula, by applying the method described in Section 6.4.1. If a state-keeping non-local loop is found, the SMT solver provides evidence, consisting of a configuration and the commands that are sent. Using this configuration, the tool determines which top bouncers are enabled and shows this to the user, in combination with the entire configuration. Multiple threads are used to speed up the process.

```
There is a state-keeping non-local loop in this system:
Racks_X2_S_X2S21 (cms_cent_dcs_01:, 141753)
 of type CMSfw_RackGeneric (39599) in state DSS_LOCK
RCA/PLC_UX55/X2S21 (cms_cent_dcs_01:, 141784)
 of type FwRackDevicePDType_109CMS (39600) in state OFF
RCA/PLC_UX55/X2S21_B_LV (cms_cent_dcs_01:, 141847)
 of type FwRackDevicePDType_104CMS (39601) in state DSS_LOCK
RCA/PLC_UX55/X2S21_A_LV (cms_cent_dcs_01:, 141818)
 of type FwRackDevicePDType_104CMS (39601) in state DSS_LOCK
In this stable configuration, a top bouncer is enabled.
If no leaf changes state, it keeps sending commands.

Enabled top bouncers in this state-keeping non-local loop:
node Racks_X2_S_X2S21 (cms_cent_dcs_01:, 141753), state DSS_LOCK:
    when ($ANY$FwRackDevicePDType_109CMS in_state {OFF})
        do TURBINE_ON
    action: TURBINE_ON
        do TURBINE_ON $ALL$CMSfw_RackGeneric
        do TURBINE_ON $ALL$CMSfw_RackAux
        do ON $ALL$FwRackDevicePDType_109CMS
```

Table 6.4: A report of a state-keeping non-local loop, generated by our tool

If a state-keeping non-local loop is found, our tool outputs a report, of which an example is given in Table 6.4. It first gives an overview of the system structure, in our example the system consists of four nodes, and a configuration. After that, it prints the top bouncers that are enabled: the node the top bouncer is in (RACKS_X2_S_X2S21), the WHEN clause that is enabled and the ACTION clause that is called. From the given FSM code, it follows that as long as any child of type FWRACKDEVICEPDTYPE_109CMS remains in state OFF, RACKS_X2_S_X2S21 will keep sending ON commands.

6.4.4 Results

We applied state-keeping non-local loop detection on the systems as resulted after bottom bouncer reduction. The results are shown in Table 6.5. These results were computed on a blade having an Intel Xeon X5660 processor and 48GB of RAM, running a 64 bit version of Windows 7.

We tried to perform the state-keeping non-local loop checks to LHCb, two of the three systems contained no state-keeping non-local loop. When applying the checks to the third system, the satisfiability tool Yices exited without providing any output. We used the tool to generate an SMT file (3GB, 10^{57389} states) and applied Yices directly to it. Unfortunately, even the blade we used could not compute an answer within two and a half days. The run time is a single measure, giving an indication.

Table 6.4 shows an example of a reported state-keeping non-local loop our tools found in CMS. A closer look at the systems of CMS revealed that in two of the 31 systems of CMS with state-keeping non-local loops, two top bouncers

		systems with state-keeping non-local loops	run time
CMS	11-2011	31 of 45	74 sec
	01-2012	31 of 45	67 sec
	03-2012	31 of 45	69 sec
	05-2012	31 of 45	83 sec
LHCb	11-2011	0 of 2 checked	>2.5 days
ATLAS	05-2012	1 of 1	2 sec
ALICE	05-2012	13 of 15	8:41 min

Table 6.5: Results of state-keeping non-local loop detection

are enabled. In the other systems, only one.

We conclude that even though we are unable to detect general non-local loops in the systems of the experiments, the reductions combined enabled us to find state-keeping non-local loops in three out of the four main experiments of the LHC. CMS, LHCb and ALICE contain state-keeping non-local loops, LHCb remains too big to compute.

7 Conclusion

Our contributions to the project were twofold: we improved existing tools and gave developers access to them, and we introduced a new desirable property: non-local loop freeness.

We implemented our own parser of FSM and identified seven possible static semantic issues that are reported on by our tools. In all the LHC experiments, static semantic errors were found: in LHCb 68, in ATLAS 19 and in ALICE 43. As a direct result of this project, the number of static semantic errors in CMS decreased from 52 to 1.

The pairwise reachability tool was re-implemented and improved, such that developers can access it from their development environment PVSS and from the website CMS Online. We made the tool more efficient by introducing reductions that limit the parent-children combinations checked. Nodes with pairwise unreachable states varied between 2.7% in LHCb to 40% in ALICE. Moreover, reachability reduction was introduced to remove unreachable states.

The local loop detection tool was re-implemented and improved as well. In CMS, some local loops were fixed as a direct result of this project, but still 14% of the nodes having modelled behaviour in CMS contains a local loop. We proved LHCb almost completely local loop free: 0.4% of the nodes contains a local loop. In ATLAS 0.8% of the nodes contain local loops, in ALICE 15%.

We defined a new kind of loop, the non-local loop, and introduced reductions that reduce the state space that has to be traversed while searching for these non-local loops. We identified a special case of a non-local loop, the state-keeping non-local loop, for which we defined and implemented a translation to a satisfiability problem. Using this translation, we were able to check CMS, ATLAS, ALICE, and a part of LHCb and find several state-keeping non-local loops. For the other non-local loops, we proved that there are systems for which no further reduction is possible.

We integrated the verification tools in PVSS and the CMS Online website to give developers access to them. Using these integrations, developers can and did check their code for errors, local loops and pairwise reachability problems, enabling them to avoid unwanted behaviour and increasing confidence in CFSM and FSM.

Suggestions for further research Several suggestions for further research have been given before:

Extend reachability checks with command knowledge	22
Implement detection of all local loops in parent-children combinations	35
Study and implement upper bound to parent-children combinations for local loops ([Kus11])	35
Automatic problem detection from logs	40
Detect general non-local loops using bounded model checking	49
Investigate children-specific top bouncer reduction	60
Support systems with multiple sources in mCRL2 translation	77

Besides these, we suggest further research or implementation into the following fields:

1. Several subdetectors use their own design guidelines. It would be interesting to investigate whether violations of these guidelines can be detected and whether the guidelines can be used to reduce state spaces.

2. Experts can take control of any node in the production system, thereby disconnecting the node from its parents, making it a source, and transferring control to the expert. This is done in emergency situations for example. However, this changes the structure of the system and can therefore affect its behaviour. It would be interesting to research the effects of these takeovers.

3. Local loops can be detected using the tools described in this report. In some cases, repairing of the CFSM class is easy. For instance, an \$ANY\$ might need to be replaced with \$ALL\$. An interesting field of further study would be to investigate whether it is possible to propose the developer a solution to eliminate the local loop, such that the impact of the solution is minimal and such that it does not introduce new local loops.

4. The detection tools for non-local loops that are described in this report only work on complete systems. It is not possible to check a subsystem for non-local loops and conclude anything about the complete system. An interesting field of further research would be to find an efficient method or approximation to check whether a non-local loop in a subsystem is present in the complete system and to check whether a non-local loop free subsystem is non-local loop free in the complete system.

Acknowledgements We wish to thank Tim Willemse, Frank Glege, Robert Gomez-Reino Garrido and Jeroen Keiren for their supervision of the project. We also thank Lorenzo Masetti, Yi Ling Hwong (former member) and Giovanni Polese from the CERN CMS DAQ group for their support and advice. We thank Clara Gaspar from the LHCb experiment, Stefan Schlenker from the ATLAS experiment and Ombretta Pinazza from the ALICE experiment for providing the CFSMs of their DCSes and for providing feedback on the results of the verification tools. We thank Dennis Schunselaar and Kevin van der Pol for their input on this report.

Bibliography

[BCC$^+$03] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58:118–149, 2003.

[BvDH$^+$01] M. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In Reinhard Wilhelm, editor, *Proc. of Compiler Construction*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

[DMdM06] B. Dutertre and L. Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[FG05] B. Franek and C. Gaspar. SMI++ object-oriented framework for designing and implementing distributed control systems. *IEEE Transactions on Nuclear Science*, 52(4):891–895, 2005.

[Fra11] B. Franek. Condition processing. private communication, 2011.

[GKM$^+$08] J.F. Groote, J.J.A. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. van Weerdenburg, J.W. Wesselink, T.A.C. Willemse, and J. van der Wulp. The mCRL2 toolset. In *WASDeTT 2008*, 2008.

[GMR$^+$09] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van Weerdenburg. Analysis of distributed systems with mCRL2. In *Process Algebra for Parallel and Distributed Processing*, pages 99–128. Chapman Hall, 2009.

[HGBGS05] O. Holme, M. González-Berges, P. Golonka, and S. Schmeling. The JCOP Framework. Technical Report CERN-OPEN-2005-027, CERN, Geneva, Sep 2005.

[HKK$^+$11] Y.L. Hwong, J.J.A. Keiren, V.J.J. Kusters, S.J.J. Leemans, and T.A.C. Willemse. Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider. Submitted for publication, 2011.

[HKW12] Y.L. Hwong, V.J.J. Kusters, and T.A.C. Willemse. Analysing the control software of the compact muon solenoid experiment at the large hadron collider. In *Fundamentals of Software Engineering*, volume 7141 of *Lecture Notes in Computer Science*, pages 174–189. Springer Berlin / Heidelberg, 2012.

[HWK$^+$11] Y.L. Hwong, T.A.C. Willemse, V.J.J. Kusters, G. Bauer, B. Beccati, U. Behrens, K. Biery, O. Bouffet, J. Branson, S. Bukowiec, et al. An analysis of the control hierarchy modelling of the cms detector control system. In *Journal of Physics: Conference Series*, volume 331, page 022010. IOP Publishing, 2011.

[Kus11] V.J.J. Kusters. Upper bound for the amount of children in the endless loop checker. private communication, 2011.

[MHS03] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *CWI. Software Engineering [SEN]*, (E 0309):1–36, 2003.

[Sco96] R. Scowen. International standard(iso 14977) extended bnf, 1996.

A mCRL2 translation

The *micro Common Representation Language 2* (mCRL2) [GMR⁺09] process algebra is used to describe concurrent systems and properties on those systems. In order to formalise CFSM and SML, in earlier projects [HWK⁺11] [HKK⁺11] [HKW12] an automatic translation of CFSM systems to mCRL2 processes was described. Using the mCRL2 *toolset*,[1] [GKM⁺08] a mCRL2 specification can be translated to a labelled transition system. Moreover, the mCRL2 toolset offers several tools to reduce those transition systems and to check whether certain properties hold on the systems.

During this project, language constructs were added that were not in the translation before and some were improved, in close collaboration with others:

- At start of this project, the translation tool had to be supplied with a manually written CFSM system structure, which made it difficult and labour intensive to apply it to large systems. This step was automated, making use of the libraries written for the other verification tools. Goal of this was to enable a user, and ourselves, to obtain a correct mCRL2 translation by only calling this tool and not having to enter anything by hand.

- The translation was designed for tree structured systems. This was changed to support all directed acyclic graphs having a single source. Further study is needed to support directed acyclic graphs having multiple sources.

- A grandparent was added, modelling operators or other control systems that can send commands to the sources at any time.

- Several ASF+SDF traversal functions were not complete. They would not collect all things, like states or commands, they were supposed to collect. This manifested itself when, for instance, a command was sent by a node, but its children did not have this command defined. As the traversal functions did not discover this command, the resulting mCRL2 model would not have this command defined.

- DO referrers were not translated correctly: the actions they pointed to could not be executed. This was fixed by pointing the program counter to the statements directly.

- Support for DO statements with parameters were added. The translation ignores the parameters as they have no useful semantics for our purposes.

[1] `http://www.mCRL2.org`

- A child is busy if it has received a command but has not responded yet with a state update message. IF statements are supposed to wait until all children that are mentioned in their guard are not busy. To verify, this behaviour was tested and confirmed in the simulation environment. The implementation was such that a node would wait for all its children. This was fixed.

- Support for the WAIT statement was added, implementation is similar to the waiting part of the IF statement.

- The NOT operator was not supported in expressions, support was added.

- The SLEEP statement was not supported and delays execution of a statement block, but has influence on neither the behaviour nor the order in which steps of the system can happen. Therefore it is translated as an mCRL2 action without further semantics.

- mCRL2 does not support all characters that CFSM allows in its names. The translation translates these unsupported characters to mCRL2-safe ones by using an encoding.

- In the translation of a MOVE_TO statement, `phase = whenPhase` was missing in the self-invocation, causing false deadlocks. This was fixed.

- As described earlier, nodes drop commands if the queue to store them is full. In the translation, for sake of feasibility of the analyses, we abstract from queue sizes greater than one. This implies that during certain loops nodes cannot receive commands. This behaviour was verified to conform to reality.

- The possibility of targeting subclasses in guards was added. For example: the CFSM class CL__&SUB can be targeted in a guard by targeting either CL or CL__&SUB. This is implemented in the Python tool that calls the ASF+SDF translation instead of in the ASF+SDF translation itself.

B System structure images

Figure B.1 shows the detector control system structure of LHCb as of November 2011, Figure B.2 of ATLAS as of May 2012 and Figure B.3 of ALICE as of May 2012. These images were generated using software written by Sjoerd Cranen of the Eindhoven University of Technology.

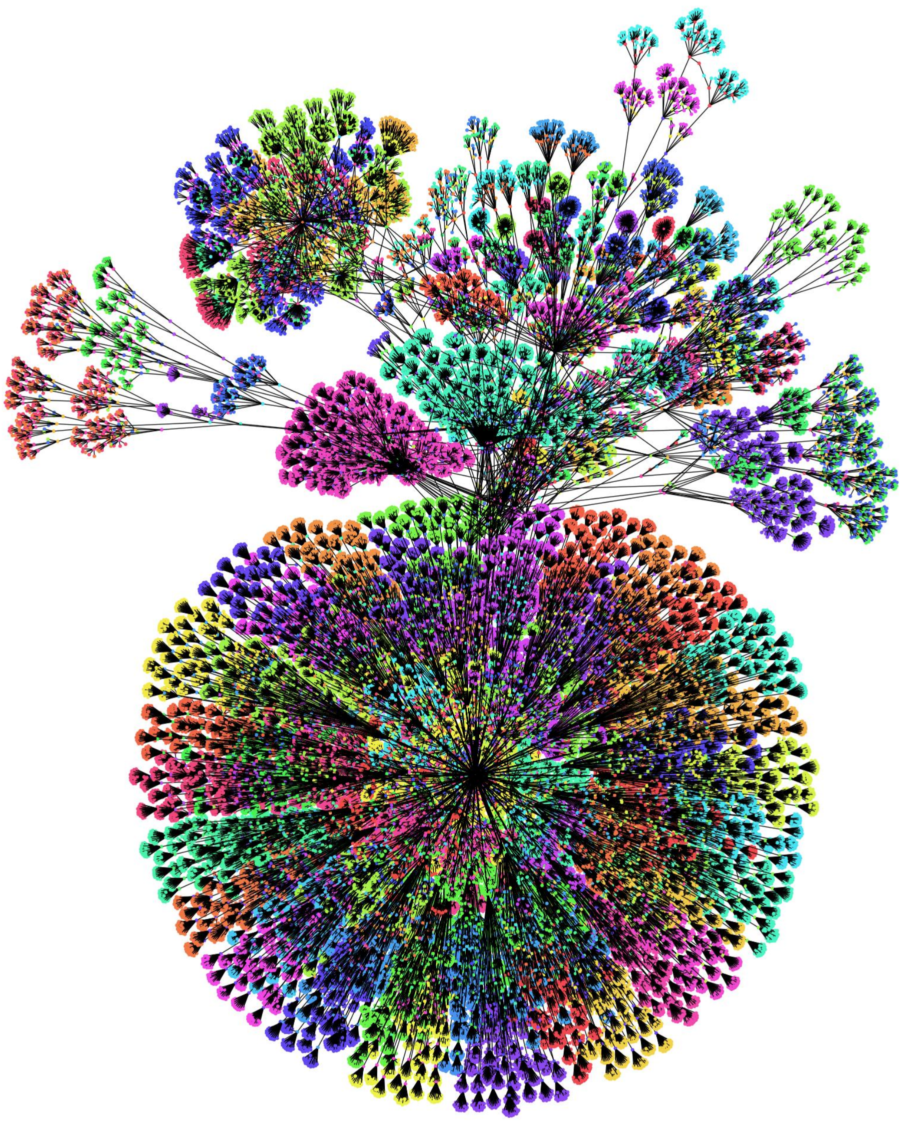


Figure B.1: Detector control system structure of LHCb (image generated using software written by Sjoerd Cranen of the Eindhoven University of Technology)

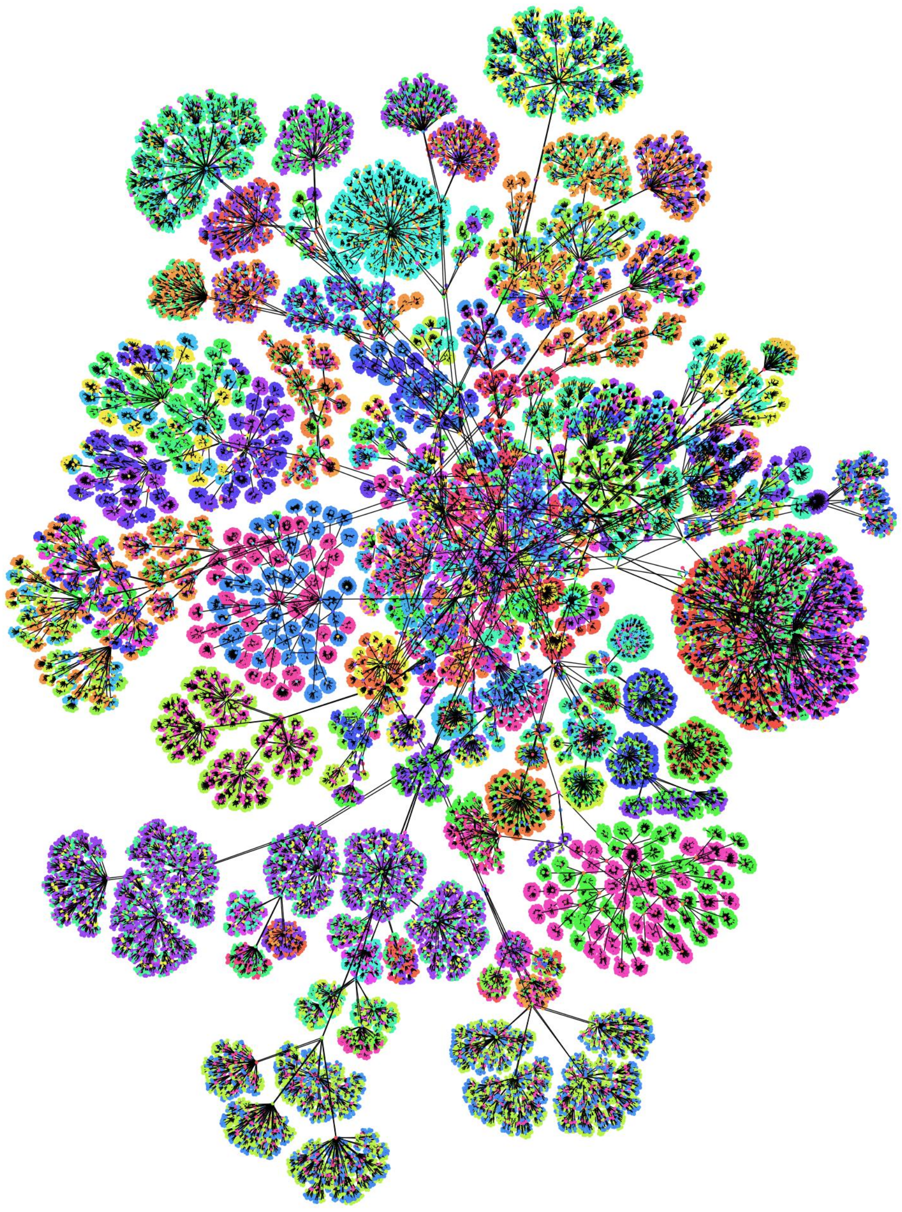


Figure B.2: Detector control system structure of ATLAS (image generated using software written by Sjoerd Cranen of the Eindhoven University of Technology)

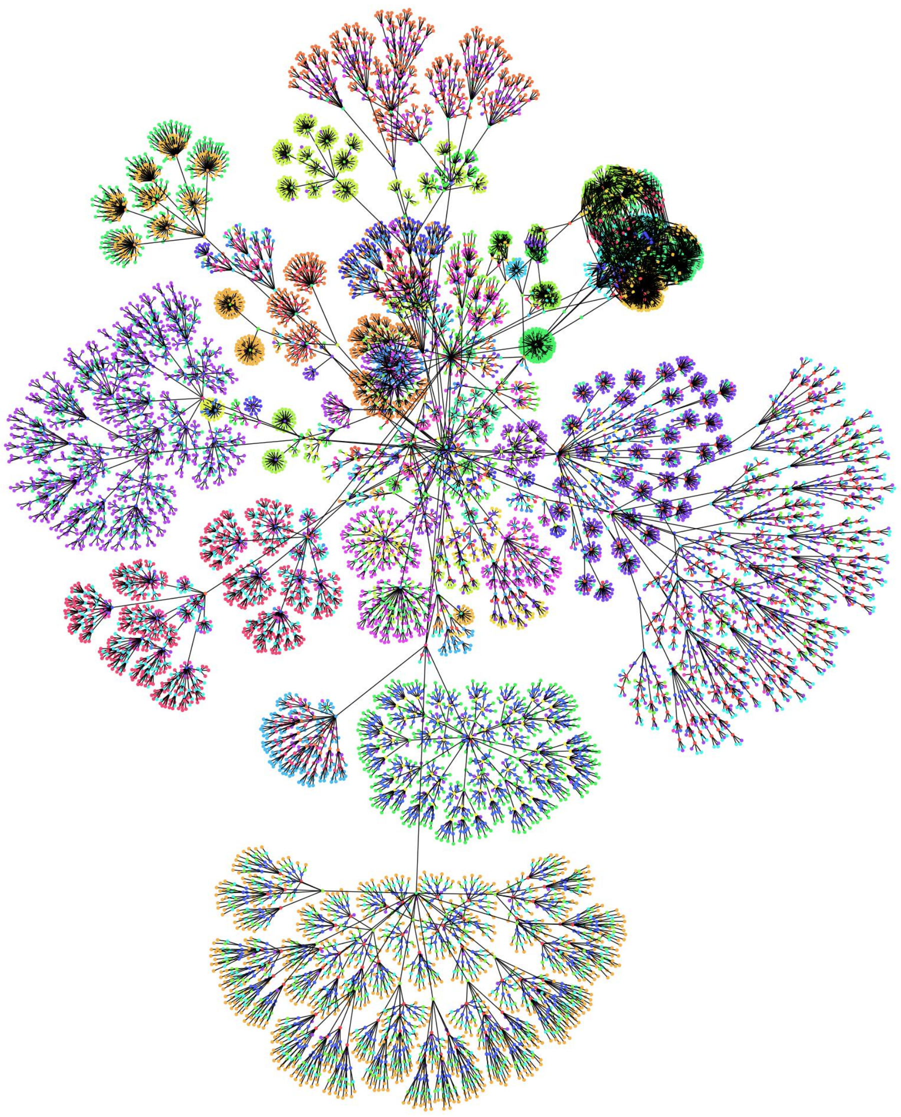


Figure B.3: Detector control system structure of ALICE (image generated using software written by Sjoerd Cranen of the Eindhoven University of Technology)

C FSM syntax and parser

This appendix gives the Extended Backus-Naur Form (EBNF)[Sco96] grammar of the FSM syntax relevant for this report. White space and comments are left out of this grammar but are allowed at every comma (,) in the grammar, except if indicated otherwise. Comments in FSM are embraced with an exclamation mark (!) and a line end. The start symbol is `specification`.

```
   specification = class, {class};
           class = 'class:', '$FWPART_$TOP$', identifier, state,
                   {state};
           state = 'state:', identifier, {when clause},
                   {action clause};
     when clause = 'when', expression, referrer;
      expression = (paren expression | not expression |
                   base expression),
                   [and expression | or expression];
 base expression = child pattern, ( 'empty' |
                   ( state operator, state specification ) );
  state operator = 'in_state' | 'not_in_state';
  not expression = 'not', '(', expression, ')';
paren expression = '(', expression, ')';
  and expression = 'and', expression;
   or expression = 'or', expression;
   child pattern = '$', ['all$' | 'any$'], identifier;
state specification =
                   identifier |
                   ('{', {identifier, ','}, identifier, '}');
        referrer = referrer do | referrer move_to |
                   referrer stay_in_state;
     referrer do = 'do', identifier;
referrer move_to = 'move_to', identifier;
referrer stay_in_state =
                   'stay_in_state', [identifier];
   action clause = 'action:', [action parameter], identifier,
                   statement, {statement};
action parameter = '(', [ action parameter single, {',',
                   action parameter single} ], ')';
action parameter single =
                   'string', identifier, '=', string literal;
  string literal = '"', {? any charachter but " ?}, '"';
```

```
        statement = statement do | statement move_to |
                    statement if | statement set |
                    statement wait | statement sleep;
     statement do = 'do', identifier,
                    ['(', statement parameter, ')'],
                    child pattern;
statement parameter =
                    ['string'], identifier, '=',
                    (string literal | identifier |
                    statement parameter object);
statement parameter object =
                    '$', identifier, {'.', identifier};
statement move_to =
                    'move_to', identifier;
     statement if = 'if', expression, 'then',
                    statement, {statement},
                    ['else', statement, {statement}], 'endif';
    statement set = 'set', statement parameter;
   statement wait = 'wait', '(',
                    child pattern, {',', child pattern}, ')';
  statement sleep = 'sleep', integer;
       identifier = character, {character};
(*in an identifier, no white space or comment is allowed*)
(*after an identifier, no 'character' is allowed*)
          integer = digit, {digit | '0'};
(*in an integer, no white space or comment is allowed*)
        character = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
                    'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
                    'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' |
                    'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' |
                    'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' |
                    'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' |
                    'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' |
                    'X' | 'Y' | 'Z' | '_' | '&' | '-' | '0' |
                    digit;
            digit = '1' | '2' | '3' | '4' | '5' | '6' |
                    '7' | '8' | '9';
```

The parser parses `and` and `or` expressions left-associative.

Observe the following rule from the grammar:

```
statement stay_in_state = stay_in_state [identifier]
```

Using the CFSM classes we obtained in November 2011, we learned the `identifier` part was optional. Making it optional however, we had introduced a production rule that makes the grammar LL(2).

The follow set of `stay_in_state` is:

```
{EOF, 'state:', 'class:', 'when', 'action:', identifier}
```

where `EOF` is the end of file. As both `identifier` and `'when'` are in the follow set, an LL parser would need at least 2 tokens to determine which symbol an encountered `when` is.
We use the Picoparser package, which provides recursive descent parsers using backtracking. However, if `stay_in_state action:` is encountered, the greedy parser will parse it as `stay_in_state action` and raise a syntax error on the semicolon. To solve this, we use the differences in the follow sets:
The follow set of an `identifier` that was intended as an `identifier` is:

```
{'class:', 'state:', 'when', 'action:', EOF}
```

The follow set of an `identifier` that was not intended as an `identifier` is:

```
{':', '(', '#', '$'}
```

After parsing the `identifier`, the parser checks whether the next character is in this last set of characters and, if that is the case, it fails, backtracks, ignores the choice and parses the `stay_in_state` correctly without an `identifier`.

D Inconsistency of **ghost** rules

The GHOST rules in combination with the common Boolean reasoning axioms yield an inconsistent system, assuming CFSM operator NOT maps to the Boolean operator $\neg$, OR maps to $\vee$, and AND maps to $\wedge$. In this system, it can be shown that true equals false:

$$\text{true}$$

By the complement rule, for every y:

$$\equiv y \vee \neg y$$

By the rule of idempotence:

$$\equiv (y \vee \neg y) \wedge (y \vee \neg y)$$

If we apply this for $y = \text{GHOST}$:

$$\equiv (\text{GHOST} \vee \neg\text{GHOST}) \wedge (\text{GHOST} \vee \neg\text{GHOST})$$

Apply twice the rule that GHOST OR x is equal to x:

$$\equiv \neg\text{GHOST} \wedge \neg\text{GHOST}$$

By the rule that NOT GHOST is equal to GHOST:

$$\equiv \text{GHOST} \wedge \neg\text{GHOST}$$

By the complement rule:

$$\equiv \text{false}$$

E Verification tools environment

A parser for CFSM classes based on the ASF+SDF meta-environment [BvDH$^+$01] was constructed in earlier projects [HWK$^+$11] [HKK$^+$11] [HKW12]. This environment takes a grammar specification in the *Syntax Defnition Formalism* (SDF) and generates a parser, syntax checker and compilers. These compilers are specified in the *Algebraic Specification Formalism* (ASF). The most recent release of ASF+SDF dates from 2008 and development has been limited to bug fixes by its authors[1]. The compilers it produces are not very fast: for instance the local loop tool needs several hours when written in ASF+SDF, while written in a more general language, it takes a few minutes. Moreover, ASF+SDF does not run on the Windows operating system which is frequently used by developers. For these three reasons, we searched for a replacement of ASF+SDF. We identified the following requirements for a replacement:

- Widely used, implying a website with documentation;
- Generated code works on Windows and preferably Linux;
- Actively maintained, meaning at least an update after 2008;
- Free;
- Code written in it must be easily maintainable.

From [MHS03] and own knowledge, a list of 24 parser generators, general programming language libraries and language development systems was obtained, see Table E.2. We performed a quick selection to obtain a short list, consisting of Rascal, Stratego/XT, TXL and Picoparse. We would recommend Rascal, once it has had a production release. Therefore for now, we choose Picoparse, a parsing library for Python. Python runs on several operating systems and is actively maintained, the library Picoparse is small and uses intuitive recursive parsers, so the need for updates and documentation, which is sparse, is low. When properly written the underlying grammar is clearly visible due to a functional programming like structure, but Python does not force the structure of the code as SDF does. Therefore, this visibility and the maintainability depends on the discipline of the parser developer.

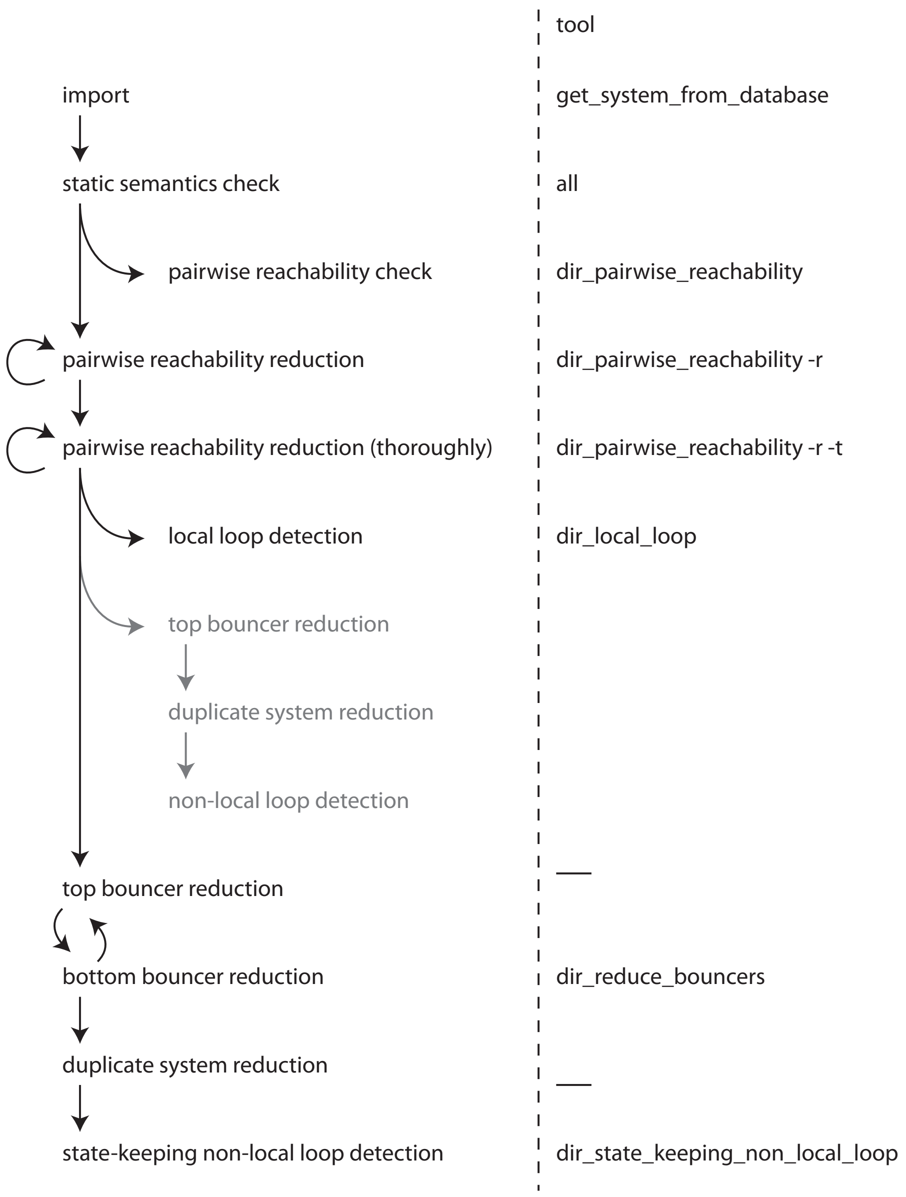
[1] `http://www.meta-environment.org/`

POPART	No website found.
LaCon	No website found.
smgn	No website found.
Kodiyak	No website found, published in 1988.
Draco	No website found, published in 1983.
Metatool	No website found, said to be commercial product. `http://craigc.com/cs/resume.html`
InfoWiz	No website found, but patented and assumed commercial product. `http://www.patentgenius.com/patent/6425119.html`
JTS	No website found, not even a reference on authors' website. `http://www.cs.utexas.edu/users/schwartz/`
Khepera	Only available through ftp site, no website found
Sprint	Not available for download: `http://phoenix.labri.fr/wiki/doku.php?do=export_xhtml&id=sprint`
Gem-Mex	Last update in 2003. `http://www.tik.ee.ethz.ch/~montages`
LISA	Last update in 2006. `http://labraj.uni-mb.si/lisa/index.html`
SPARK	Last update in 2002. `http://pages.cpsc.ucalgary.ca/~aycock/spark/`
DMS	Commercial product. `http://www.semdesigns.com/products/DMS/DMSToolkit.html`
metafront	Only a prototype is available, no documentation. References list updated up to 2008. `http://www.brics.dk/metafront/`
AsmL	Not platform independent. `http://research.microsoft.com/en-us/projects/asml/`
ASF+SDF	Not platform independent, maintainance ended. `http://www.meta-environment.org/`
Eli	Consists of C code that does not run on Windows. Generated code is said to be 'highly portable', but the FAQ suggest the authors did not even test this themselves. `http://eli-project.sourceforge.net/elionline`
pure Python	Too general, Picoparse and SPARK are Python libraries more dedicated to parsing.
SmartTools	Not a parser generator, but a SOA oriented helper tool. `http://www-sop.inria.fr/members/Didier.Parigot/SmartTools/eclipse/documentation/`
RascalMPL	On shortlist. We suggest to use RASCAL if a production version is released. `http://www.rascal-mpl.org/`
Stratego/XT	On shortlist. Too buggy: crashed Eclipse often and produced invalid C-code. `http://strategoxt.org/`
Picoparse	On shortlist. `https://github.com/brehaut/picoparse/`
TXL	On shortlist. Requires a single grammar for all intermediate results. `http://www.txl.ca/`

Table E.2: Verification tools environments

F Tool chain

Figure F.1 shows the chain of reductions as we applied it. The grayed out part of the chain was not implemented by us. The right column shows what tool to call to perform a task.



tool
import
get_system_from_database
static semantics check
all
pairwise reachability check
dir_pairwise_reachability
pairwise reachability reduction
dir_pairwise_reachability -r
pairwise reachability reduction (thoroughly)
dir_pairwise_reachability -r -t
local loop detection
dir_local_loop
top bouncer reduction
duplicate system reduction
non-local loop detection
top bouncer reduction
bottom bouncer reduction
dir_reduce_bouncers
duplicate system reduction
state-keeping non-local loop detection
dir_state_keeping_non_local_loop

Figure F.1: Chain of tools and reductions

G Local loop in production

In this appendix, a selection of a log file of the CMS system is shown. This log shows that a node with name PIXELBARREL_BMI_S7 looped through states ANALOG_ON_RED and LVMIXED for about 15 minutes, before one of its children changed state to ON_LV. Section G.2 shows the loop report of the corresponding CFSM class.

G.1 Log

```
Sun Nov 06 15:23:57 2011 - [PIXELBARREL_BMI_S7]
                                      in state [LVMIXED]
Sun Nov 06 15:23:58 2011 - [PIXELBARREL_BMI_S7]
                                      in state [ANALOG_ON_RED]
Sun Nov 06 15:23:58 2011 - [PIXELBARREL_BMI_S7]
                                      in state [LVMIXED]
Sun Nov 06 15:23:58 2011 - [PIXELBARREL_BMI_S7]
                                      in state [ANALOG_ON_RED]
```

... 42 times per second ...

```
Sun Nov 06 15:38:08 2011 - [PIXELBARREL_BMI_S7]
                                      in state [LVMIXED]
Sun Nov 06 15:38:08 2011 - [CMS_TRACKER:PIXELBARREL:
    PIXELBARREL_BMI:PIXELBARREL_BMI_S7:PIXELBARREL_BMI_S7_LAY1]
                                      in state [ON_LV]
Sun Nov 06 15:38:08 2011 - [CMS_TRACKER:PIXELBARREL:
    PIXELBARREL_BMI:PIXELBARREL_BMI_S7:PIXELBARREL_BMI_S7_LAY3]
                                      in state [ON_LV]
Sun Nov 06 15:38:08 2011 - [PIXELBARREL_BMI_S7]
                                      in state [ANALOG_ON_RED]
Sun Nov 06 15:38:08 2011 - [PIXELBARREL_BMI_S7]
                                      in state [LVMIXED]
Sun Nov 06 15:38:08 2011 - [PIXELBARREL_BMI_S7]
                                      in state [ON_LV]
```

G.2 Report

Please note that generation of these reports was not yet finished at time of discovery, so the report is not as informative as more recent reports (node names are not displayed).

```
This parent contains a local loop:
TkControlGroup (38772)

The parent can walk through states ANALOG_ON_RED, LVMIXED and
ANALOG_ON_RED in some parent-children combinations. For instance,
the parent will loop if it has the following children in states:
TkOffEmergencySwitcher(38651) in state OK
FwCaenChannelCtrl(38804)      in state ON
TkDistinguishCg(38822)        in state OFF
TkPowerGroup(38842)           in state ANALOG_ON_RED
TkPowerGroup(38842)           in state ANALOG_ON_RED
TkPowerGroup(38842)           in state ANALOG_ON_RED
TkPowerGroup(38842)           in state ANALOG_ON_RED
TkPowerGroup(38842)           in state ANALOG_ON_RED
TkPowerGroup(38842)           in state ANALOG_ON_RED


When clauses involved in this loop:
state: ANALOG_ON_RED
    when ($ANY$TkPowerGroup not_in_state {DIGITAL_ON_RED})
        move_to LVMIXED
state: LVMIXED
    when (($ALL$FwCaenChannelCtrl in_state {ON}) and
        ($ALL$TkPowerGroup in_state {ANALOG_ON_RED}))
        move_to ANALOG_ON_RED
```

The copy-paste error was present in the first when clause, which should have been:

```
when ($ANY$TkPowerGroup not_in_state {ANALOG_ON_RED})
    move_to LVMIXED
```